TURING 图灵程序设计丛书

Microsoft

C# 敏捷开发实践

【英】Gary McLean Hall 著　许顺强 译

拥抱敏捷，编写自适应代码
轻松应对恼人的需求变更



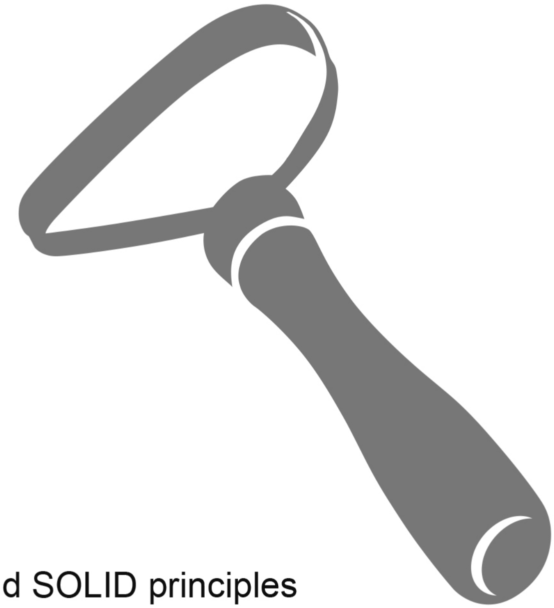
Adaptive Code via C#
Agile coding with design patterns and SOLID principles

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

C#
敏捷开发实践

【英】Gary McLean Hall 著　许顺强 译



Adaptive Code via C#

Agile coding with design patterns and SOLID principles

人民邮电出版社
北　京

图书在版编目（C I P）数据

C#敏捷开发实践 / (英) 加里·麦克莱恩·霍尔
(Gary McLean Hall) 著 ; 许顺强译. -- 北京 : 人民邮
电出版社, 2016.7
（图灵程序设计丛书）
ISBN 978-7-115-42789-2

Ⅰ. ①C… Ⅱ. ①加… ②许… Ⅲ. ①C语言—程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2016)第139459号

内 容 提 要

本书共分为敏捷基础、编写 SOLID 代码和自适应实例三大部分，将理论与实践相结合，介绍了当前使用 Microsoft .NET Framework 进行 C# 编程的最佳实践，详尽探讨了 C# 开发人员如何应用 Scrum 等敏捷方案实现高质量、自适应的代码，并给出大量代码示例，是 .NET 中高级程序员进阶的实用指南。

本书的读者对象为有一定经验的 .NET 开发人员。

◆ 著　　　[英] Gary McLean Hall
译　　　许顺强
责任编辑　朱　巍
执行编辑　杨　琳　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：800×1000　1/16
印张：22
字数：533千字　　　　2016年 7 月第 1 版
印数：1－3 000册　　　2016年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-2389号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

谨以此书献给 Amelia Rose。

译 者 序

翻译这本书算是圆了我学生时期的一个心愿：在一本纸质书的封面上印上自己的名字。作为一名 IT 技术人员，总是感觉自己的时间不够用，因为技术永无止境，让我从不敢懈怠。以前阅读过许多“大侠”翻译的不少书籍，但对于译者的辛酸和困难都没有太大的感触。直到自己开始实际动手翻译时，才发现整个过程中心情都是忐忑不安的，担心自己的翻译不能清楚传达原书作者的真实意图，害怕自己的译文会让读者觉得乏味，于是，总是会对不满意的翻译片段心生焦虑，也尽了自己最大努力对译文的选词和顺序进行反复的推敲和斟酌。当然，由于自身能力和精力有限，相信大家肯定会找出翻译中需要改进的地方，在此先行感谢大家的热心指正。

原书内容的精彩我就不多赘述。本书主要针对 C#程序员，基于敏捷方法论，介绍使用 Microsoft .NET Framework 进行 C# 编程的当前最佳实践，其中包括了从敏捷项目过程到代码编写的理论和实践的详细讲解。对于那些需要实操指导的读者来说，几乎全部内容都可以直接应用在实际敏捷项目的管理和编码活动当中。如果你是一名初学者，可以在本书中学习到使用 C# 进行敏捷开发的常见模式和实践，明辨其优劣，让自己走在正确的方向上，为后续的能力提升打下良好的基础。如果你是一名中级开发人员，可以在本书中学习到业界的最佳实践，了解各种实践组合，对 SOLID 原则获得深入的理解，并完全认识到其在实际代码开发中带来的益处。如果你是一名高级开发人员，毫无疑问，你将获益最多。本书提供了大量设计模式、SOLID 原则、单元测试、重构等理论的示例，将理论与实践关联起来，让你可以直接拿来应用于工作之中。

当然，尽信书不如无书，相信有读者会对书中所讲并不完全赞同，但表达意见的前提是首先要理解书中讲解的本意。技术皆有优劣，作为 IT 技术人员，切忌用个人主观情绪来表达对技术观点的不满。我有时会听到周围有人说：“我就是看着不爽！”他们习惯了一刀切，也不明白一句古话：“三人行，必有我师焉。”原书作者 Gary 已经在全书技术理论和实例讲解过程中很好地穿插了优缺点和应用场景的讨论。认真读完一本技术书的收获应该是：明悉技术的概念和原理，了解其优缺点，并且知道其适用场合；如果还能在实践中针对缺点提出改进，那就再好不过了。

许顺强
2016 年 3 月 21 日

前　　言

本书英文书名中首先提到的一个关键术语是*自适应代码*（Adaptive Code），这一关键术语很好地诠释了应用本书中介绍的基本原则能够达成的效果：无需大量返工，代码即可自动适应后续新的需求和无法预见的场景。本书旨在将使用 Microsoft .NET Framework 进行 C#编程的当前最佳实践囊括于一卷之中。尽管其他书中也会涵盖本书中的某些内容，但这些书要么偏重于讲解理论，要么就不是特定于.NET Framework 开发的。

编写代码不能急于求成。与阻碍变化的代码库相比，如果你的代码具有自适应能力，你就能够更加快速、轻松地对其进行更改，并且不会引发多少错误。相信大家都知道，对于需求，不变的主题就是变化。因此管理需求变更是软件项目成败的一个关键因素。面对需求变更，开发人员可以有很多种应对方法，这些方法可以归类到下面要讲述的两种方法论。这两种方法论的主旨几乎是截然相反的，特别是在变更处理的连续性上。

第一种是瀑布方法论。这种方法论要求开发人员必须遵循严格的流程。应用这种方法论的项目中，大到要遵循的开发流程，小到要实现的类型设计都很不灵活，甚至几乎与 50 年前用穿孔卡片进行编程开发一样死板。所有瀑布方法论都在竭尽全力确保软件很难被自由更改。软件开发被划分为分析、设计、实现和测试几个显著不同的阶段，而且整个过程是单向的。一旦进入实现阶段，用户就很难去变更需求，或者说至少要付出昂贵的代价才能变更需求。当然，代码也*无需*为需求变更作任何准备，因为整个瀑布流程几乎不提供任何其他选项。

第二种是敏捷方法论。它不仅仅是另一种选择，而且是对瀑布方法论的一种*彻底推翻*。敏捷流程的主旨就是拥抱变化，它被看作是客户和开发者之间的一个必要的联系纽带。如果客户想要对自己付费的产品进行某些改变，就应该把时间和资金的代价与需求的变更关联起来，而不是直接把变更加入流程中正在进行的阶段。软件工程的基础是源代码，相对于物理工程，它具有更好的可塑性。建造一座房子的过程就是使用水泥把砖逐块粘合在一起，所以改变房子设计的代价就很自然地与房子的建造完成度直接相关了。假设工程尚未开始，只有设计蓝图，那么变更设计的代价相对来说就会很低。如果已经装好了窗户，布好了电线、管线，此时再想把楼上的浴室改到楼下的厨房旁，那代价就会非常高了。软件产品的源代码具有良好的可塑性，因此移动特性和修改用户界面的导航看起来不应该有很高的代价，但不幸的是，事实并不总是如此。单单时间成本就经常不允许在软件产品中进行这样自由的变更。在我看来，这主要就是因为代码缺乏对需求变更的自适应能力。

本书将通过一些实际的例子，为大家演示和讲解敏捷流程以及如何编写自适应代码。

本书面向的读者

本书的意图是要把理论与实践关联起来。如果你是经验丰富的高级开发人员，想要找一些设计模式、SOLID 原则、单元测试、重构等理论的示例，那么这本书就是为你而作。

如果你是具有一定经验和能力的中级开发人员，想要学习业界的最佳实践，了解它们是如何配合使用的，或者你对现在的业界最佳实践组合有疑问，都可以从这本书中获益，因为现实的项目开发中很难找到简单且容易理解的实例或者理论。开发人员对大多数 SOLID 原则已经有所了解，但是对于其中比较复杂的开放与封闭原则（第 6 章）和 Liskov 替换原则（第 7 章）的理解还不够充分。即使是经验丰富的开发人员，有时也无法完全认识到依赖注入（第 9 章）给代码开发带来的好处。与此类似，接口（第 3 章）能给代码带来的适配灵活性也经常被忽视。

如果你是刚刚入门的初级开发人员，读完这本书，也会有所收获。你可以学习到常见的模式和实践，并且知道哪些方面是好的，哪些方面从长期来看是不好的。我见到的软件开发实习生所写的代码有很多共同点，到处都可以看到随从反模式（第 2 章）和服务定位器反模式（第 9 章）的代码。通常，实习生已经具备了很多方面的软件开发技能，要把他变成一个重量级的开发人员，只需要在正确的方向上推他们一把即可。本书也提供了多种可选的实践方案，并对其优缺点进行了详细解释。

阅读本书的前提条件

要阅读这本书，你应该具备一些在语法上与 C#类似的编程语言（比如 Java 或 C++）的实战经验，也应该精通条件分支、循环和表达式等核心过程编程概念。此外，还应该有使用类进行面向对象开发的经验，并且对接口的概念有所了解。

本书不适合哪些人

如果你刚刚开始学习编程开发，那么本书并不适合你，因为书中涉及一些高级开发话题，需要你对基本的编程开发概念有深入的理解。

本书结构

本书共分为三个部分，每一部分都以上个部分为基础。尽管如此，也可以从任何一个部分开始阅读本书。每个章节都详细讲解了一个完整的主题，并在适当的地方包括了指向其他章节的交叉索引。

第一部分　敏捷基础

这个部分会讲解如何以自适应方式开发软件的基础概念，其中包括业界有名的敏捷流程 Scrum，该流程要求代码具有自适应变更的能力。这一部分的所有章节都围绕接口、设计模式、

重构和单元测试进行详细的讲解。

- 第 1 章　Scrum 介绍

  这一章是本书的开篇，首先介绍一种业界知名的敏捷项目管理方法论 Scrum，然后详细介绍 Scrum 项目中相关的工件、角色、度量标准和阶段的概念，最后为大家展示如何在敏捷环境下组织资源和代码。

- 第 2 章　依赖和分层

  这一章将引领你一起探索依赖和架构分层。代码要做到自适应，前提是解决方案的结构允许这样做。首先讲解三种不同类型的依赖：第一方、第三方和框架；然后讲解如何从反模式（应该避免的）到模式（应该使用的）来管理和组织所有的依赖关系；最后会介绍一些高级主题供你进一步阅读，比如面向切面编程和非对称分层等。

- 第 3 章　接口和设计模式

  在现代.NET 应用的开发中，接口几乎无处不在，但是它们也经常被滥用、误解和错用。这一章将首先通过一些常见和实用的设计模式来展示接口的多种用途；然后阐明除了使用接口进行简单的抽象外，还能够以多种不同的方式使用接口来解决同一个问题。如果能够流利地使用开发者武器库中的混合类型、鸭子类型和流接口，将更能体会到接口的强大用途。

- 第 4 章　单元测试和重构

  单元测试和重构正在成为开发人员必备的两个实战技能，只有同时应用这两个技能，才能编写出自适应代码。没有完备的单元测试，重构动作肯定会造成很多错误；没有重构，代码则会变得臃肿、僵化且难以理解。这一章会从一个十分简单的单元测试示例开始讲解，然后扩展讲解更高级、实用的模式和实践，比如流断言、测试驱动开发和模拟。本章还提供了真实的重构示例来讲解如何改善源代码的可读性和可维护性。

第二部分　编写 SOLID 代码

这一部分以第一部分为基础，每一章都会专门讲解 SOLID 中的一个原则。这些章节不会单单讲解为什么要使用这些原则，还有详细的实战示例来讲解如何在编码实现中使用这些原则。每一章都会提供一个实际项目中的例子来展示 SOLID 原则的作用。

- 第 5 章　单一职责原则

  这一章会为开发人员展示如何使用修饰器和适配器模式来实践单一职责原则。应用这个原则后，类的数目会增加，但是每个类的规模会变小。相对于那些功能繁多的大型类，小型类可集中解决一个大型问题中的一小部分。聚合使用这些小型类会比使用单个大型类更强大、更灵活。

- 第 6 章　开放与封闭原则

  这一章要讲解的是开放与封闭原则，它的概念非常简单，但是它对代码有着举足轻重的影响。它负责确保那些遵守所有 SOLID 原则的代码不会被改动，而只是添加新代码。这

一章还会讨论跟开放与封闭原则相关的可预测变化的概念，并且会探讨它如何能够识别后续自适应扩展的切入点。

- 第 7 章　Liskov 替换原则

  这一章会展示在代码中应用 Liskov 替换原则所带来的好处，特别要提到的是，Liskov 替换原则有助于确保开放与封闭原则的应用，并避免代码改动所带来的副作用。这一章还会介绍代码契约，它包括前置条件、后置条件和数据不变式三个要素，可以通过代码契约工具来确保代码满足它们。此外，这一章还描述了一些子类型原则，比如协变、逆变和不变性原则，并介绍了违背这些原则会带来的不良影响。

- 第 8 章　接口分离原则

  在实际的代码中，接口和类的问题类似，它们的规模通常太过庞大。接口分离原则是一个经常被忽视却简单有效的实践原则。这一章会展示将接口规模限制到足够小，以及配合使用这些小型接口能够带来的好处。此外还会探究可能会促使接口分离的各种不同的原因，比如客户端需求和架构需求。

- 第 9 章　依赖注入原则

  这一章的核心是依赖注入，它能够将本书中讲解到的所有特性结合为一个整体。依赖注入真的非常重要，没有它，很多其他原则都无法起作用。这一章会详细介绍依赖注入并比较实现依赖注入的不同方法。这一章还会讨论如何使用控制反转容器来管理对象的生命周期，以避免服务定位器等反模式；此外，还会讨论如何识别组合根和解析根。

第三部分　自适应实例

这一部分以一个示例应用为主线，将本书的剩余内容组织到一起。尽管这些章节中有很多代码，但我也提供了很丰富的注释和讲解。因为本书的讲解背景是敏捷环境，所以下面这些章节会按照 Scrum 的冲刺进行组织，并且所有工作都是由积压工作项和客户变更请求而来。

- 第 10 章　自适应实例简介

  这个部分要实际开发的示例应用是一个基于 ASP.NET MVC 5 开发的在线聊天应用。这一章会先详细描述这个应用，并给出一个简要的设计作为后续架构的指导，最后还给出了产品积压工作上所有特性的解释。

- 第 11 章　自适应实例冲刺 1

  这一章介绍如何使用测试驱动开发方法来开发示例应用的首要特性，包括查看和创建聊天室和消息。

- 第 12 章　自适应实例冲刺 2

  这一章讲解客户对应用提出需求变更，以及整个开发团队如何通过自适应代码来适应这些必然会发生的变更。

附录　自适应工具

附录简要介绍如何使用 Git 源代码控制从 GitHub 上下载代码，以及如何使用 Microsoft Visual Studio 2013 编译下载的代码。请注意，附录不是完整的 Git 使用说明，你可以在网上找到很多非常详尽的资料，比如 Git 的官方教程：http://git-scm.com/docs/gittutorial。

通过快速 Web 搜索可以找到其他来源。

附录还会简要介绍其他的一些开发工具，比如持续集成和开发环境。

本书约定

本书中有若干反复出现的约定用法，你可以在微软出版社的出版物中找到标准的解释，我也在这里先给出一些简要的解释。

代码清单

书中代码清单会在适当的地方出现，相关的代码会在同样的背景块中。比如下面的代码清单 I-1。

代码清单 I-1　这是一个代码清单，在书中会经常出现

```
public void MyService : IService
{

}
```

只要看到代码清单，你都应该关注其中特定的一部分代码。比如，当对上一份示例代码进行改动时，与改动相关的代码就会加粗显示。

阅读辅助和补充信息

阅读辅助主要提供一些与主题相关的边栏，比如注意或者警告，而补充信息则是提供一些更进一步扩展主题的信息。下面是一些示例。

> **注意**　这是阅读辅助，其中包括与主要内容相关的小信息，不过具有额外的重要性。

> **这是补充信息**
>
> 尽管已经尽量缩短篇幅，但是补充信息通常包含与主要话题不太相关的较长讨论。

图片

有时候，无论文字解释有多么形象，还是不足以表达确切的含义。这时候，就必须要提供图片了。书中所有使用 Microsoft Visio 2013 创建的图表都是黑白色的，目的是为用户提供清晰的说明。同样，截图也是在高对比度主题下获取的。

系统要求

为了使用书中提供的代码示例，需要以下硬件和软件。

- 安装了以下任意一个操作系统：Windows XP SP3（Starter Edition 除外）、Windows Vista SP2（Starter Edition 除外）、Windows 7、Windows Server 2003 SP2、Windows Server 2003 R2、Windows Server 2008 SP2 以及 Windows Server 2008 R2。
- 安装了 Visual Studio 2013 的任意版本（如果你使用的是 Express Edition 系列产品，可能需要下载几个安装包）。
- 安装了 Microsoft SQL Server 2008 Express Edition 或者更高版本（2008 或 R2 版本），以及 SQL Server Management Studio 2008 Express 或更高版本（Visual Studio 安装包中已经包括，Express Edition 则需要单独下载）。
- 处理器频率不低于 1.6 GHz（推荐 2 GHz）。
- 内存大小不低于 1 GB（32 位操作系统）或 2 GB（64 位操作系统）。如果运行虚拟机系统或者使用 SQL Server Express 版本，则另外需要 512 MB 内存，更高的 SQL Server 版本需要更多内存。
- 硬盘可用空间不低于 3.5 GB。
- 硬盘转速不低于每秒 5400 转。
- 显卡必须支持 DirectX 9，并且支持 1024×768 或者更高分辨率的显示。
- DVD 光驱（如果需要从 DVD 安装 Visual Studio）。
- 能够连接互联网以便下载软件或者代码示例。

基于不同的 Windows 配置，你可能需要本地管理员权限才能安装或配置 Visual Studio 2013 和 SQL Server 2008 等产品。

下载示例代码

我会尽力保证书中的代码片段都是一个可独立运行的应用或者单元测试中的一部分。我使用 MSTest 写了很多简单的单元测试，因此无需使用其他额外的测试器，另外我也使用 NUnit 写了一些更复杂的单元测试。我使用 Visual Studio 2013 Ultimate 编写了书中所有的代码。尽管一些代码是用 Visual Studio 2013 Ultimate 预览版编写的，但是它们都能够使用正式版成功编译和测试。我尽量不使用 Visual Studio 2013 Express 版本之外的特性，但是有些主题的代码必须使用，因此你可能需要安装一个付费版本来运行部分代码。

代码可以从 GitHub 下载，网址是：http://aka.ms/AdaptiveCode_CodeSamples。

附录包含了 Git 的使用方法简介。

我很乐意看到你的留言，下面是我的 WordPress 博客网址：http://garymcleanhall.wordpress.com。

致谢

本书的作者署名只有我自己，这肯定是不准确的。因为如果没有大家给我提供各方面的帮助，我根本无法完成这本书。

感谢我的妻子 Victoria，她是这本书成为可能的关键。这不是空话，这就是事实。

感谢我的女儿 Amelia，她每天的表现都十分完美。

感谢我的母亲 Pam，她帮助我校对，感谢她对我毫无保留的鼓励。

感谢我的父亲 Les，感谢他的所有付出。

感谢我的兄弟 Darryn，感谢他给我源源不断的指导。

感谢来自 Online Training Solution 公司的 Kathy Krause，她的努力让这本书读起来舒服流畅。

感谢 Devon Musgrave 的耐心帮助。

勘误、更新和相关支持

我们已经尽了最大的努力来确保书中内容以及相关材料是正确的。你可以从下面的网址得到本书的更新，其中包括完整的勘误表：http://aka.ms/Adaptive/errata。

如果你发现了勘误表之外的新错误或者不当之处，也请在上面的网址提交。

如果你需要额外的支持，可以发送电子邮件给微软出版社支持中心：mspinput@microsoft.com。

此外，有关微软软件或者硬件产品的支持不能通过上面的网址获得，请访问以下网址：http://support.microsoft.com。

微软出版社的免费电子书

微软出版社有很多免费的电子书，深入讲述了很多技术主题。这些电子书以 PDF、EPUB 和 Mobi（Kindle 电子书阅读器支持的格式）等格式供读者下载：http://aka.ms/mspressfree。

常去看看，你会发现很多新的电子书。

期待你的反馈

在微软出版社看来，读者的满意度始终是第一位的，读者的反馈是最有价值的资产。请在下面的网址将你对这本书的看法提交给我们：http://aka.ms/tellpress。

我们知道大家都很忙，所以尽量只设计了少量的简单问题。大家的回答会被直接发送给微软出版社的编辑（不需要个人信息）。谢谢大家的热心反馈。

保持联系

让我们保持联系，下面是我们的 Twitter 网址：http://twitter.com/MicrosoftPress。

电子书

扫描如下二维码，即可购买本书电子版。

目　录

Part 1 第一部分

敏捷基础

本部分内容

这一部分主要介绍敏捷原则和实践的基础知识。

编写代码是软件开发的核心工作，而编写好用的代码有很多不同的方式。即使抛开平台、语言和框架的影响，对于一个开发人员，最简单的一个功能的实现也会有多种选择。

在软件开发产业，开发成功的软件产品一直以来都是焦点。但是近几年，开发人员开始重视那些能够被重用并能提高代码质量的实现模式和实践，因为大家逐渐意识到软件产品的质量是无法与代码的质量割裂开来的。随着时间的推移，质量差的代码会逐渐降低产品的质量，至少一定会延迟可工作软件的完整交付。

为了开发高质量的软件产品，开发人员必须努力确保编写的代码是可维护的、可读的，并且是经过测试的。在此基础上，对开发人员提出了一个新的要求：编写的代码也应该具备一定的自适应变更的能力。

这一部分的四个章节主要介绍现代的软件开发流程和实践。这些流程和实践有一个统一的类型名称，那就是敏捷，以表达它们具有快速响应变更和改变方向的能力。敏捷流程（Agile Process）给软件开发团队推荐了很多的方法，用于快速得到反馈、响应并调整工作焦点。敏捷实践（Agile Practice）还推荐了很多方法来帮助开发团队编写出自适应代码。

第 1 章 Scrum介绍

完成本章学习之后，你将学到以下技能。

- 给项目的主要干系人分配角色。
- 识别Scrum需要生成的各种文档和其他工件。
- 监测Scrum项目进度。
- 诊断Scrum项目的问题并提出补救措施。
- 高效主持Scrum会议以达成最大的会议成果。
- 对比Scrum和其他敏捷或严苛方法之间的优劣。

Scrum是一个具体的项目管理方法论。更准确地说，它是敏捷方法的一种。Scrum的核心概念是以迭代的方式为软件产品增加价值。整个Scrum流程是可重复的，也可以迭代多次，一直持续到整个产品完成或者流程被终止。这些迭代被称为*冲刺*（sprint）。经过若干个冲刺得到的软件是随时可以发布的。整个软件产品的所有工作项都会在*产品积压工作*（product backlog）上按照优先级排列，在每个冲刺开始时，开发团队会把在这个新的冲刺中承诺要完成的工作项添加到*冲刺积压工作*（sprint backlog）上。Scrum中工作项的单位是*故事*（story）。产品积压工作实际上就是一个排好序的候选故事队列，每个冲刺则由要在这一个冲刺中承诺要开发完成的故事组成。图1-1展示了Scrum流程的框架。

Scrum过程中，开发团队内部或外部相关角色人员会产出一些文档工件，此外他们还会一起参加一些会议。只用一章的篇幅当然无法从项目管理的角度完整展示整个Scrum的细节，但是本章会尽量提供足够多的细节，为你进一步深入学习Scrum打下基础，并为后续的每日实践提供一个方向性的指导。

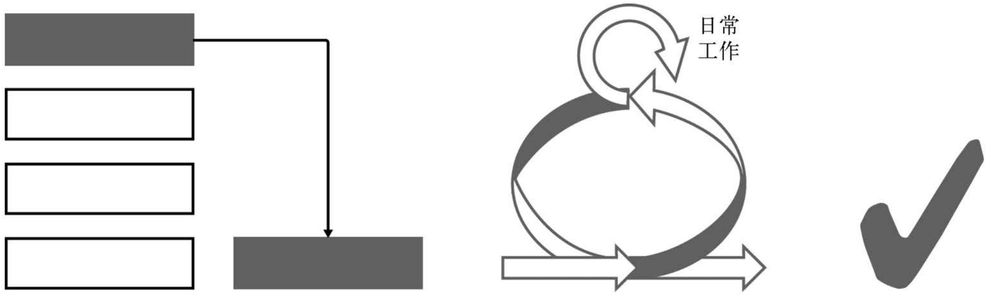


图1-1　Scrum的工作方式看起来就像一条为软件产品逐步添加小特性的生产线

> **Scrum是一种敏捷方法**
>
> **敏捷方法论**（Agile）包括一组轻量级软件开发方法，它们都允许及时响应客户的需求，即使项目已经在进行过程中了。敏捷是在很多严苛的结构化项目实践失败的教训中应运而生的。敏捷宣言列出了敏捷和严苛方法论之间详细的对比项。大家可以在以下网址查看完整的敏捷宣言：http://www.agilemanifesto.org。
>
> 敏捷宣言最初只是由十七位开发人员联名发起的。但是从那一刻起，敏捷方法的影响力就在日益扩展，现在已经成为了敏捷大环境下各个软件开发角色公认的基本约定。Scrum是目前敏捷方法论中应用最广泛、最常见的一个流程实现。

1.1　Scrum 与瀑布

根据我多年的软件产品开发经验，敏捷方法比瀑布方法工作的效果要好，而且我也乐于推广各种敏捷流程。瀑布方法论的根本问题在于它过于严苛和僵化。图1-2展示了一个瀑布工程涉及的流程阶段。

从图1-2可以看到，每个阶段的输出都是下一个阶段的输入，而且每个阶段必须在进入下一个阶段前完成。这就要求在一个阶段完成时没有遗留任何错误、问题、难点或者误解，因为整个过程只是单向的。

瀑布流程还要求在任一阶段完成后不允许再发生任何更改，而这与大量的经验和统计数据不符。变化本身就是生活固有的一部分，软件工程也不例外。瀑布方法支持的这种应对变更的僵硬

态度是高代价的、不值得的，而且是肯定可以避免的。瀑布方法论认定可以花费更多时间在需求和设计阶段来识别出所有潜在的变更，这样在后续阶段就不会发生变更了。这个观点很不靠谱，因为变更总会发生，不论你愿意与否。

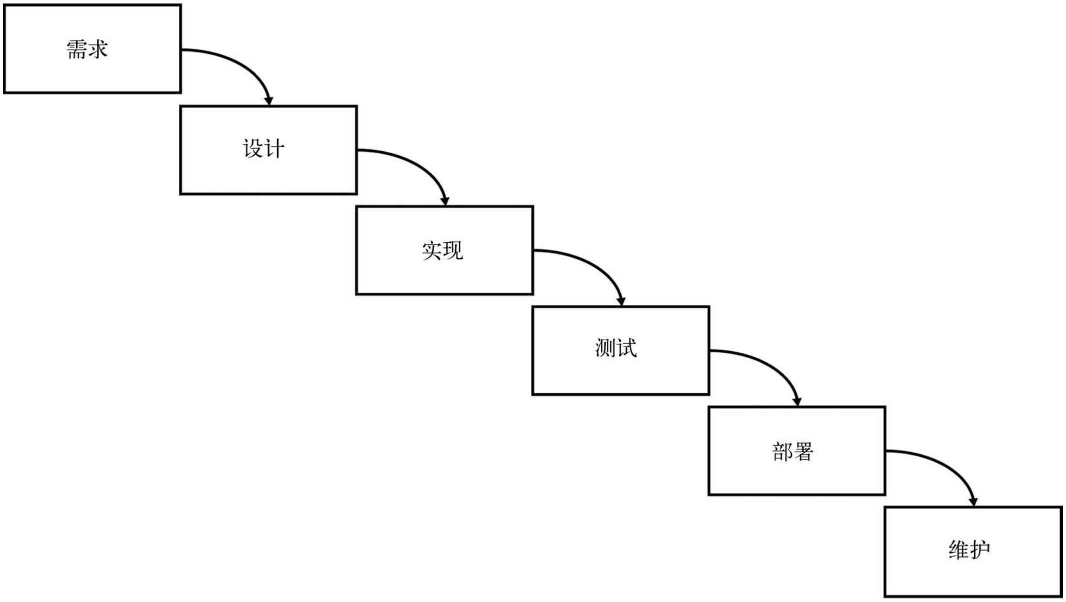


图1-2 瀑布开发流程

为了应对变更，敏捷流程引入了另外一种方法，这个方法主动拥抱变化并且允许每个人都能快速响应任何变更。尽管敏捷（包括Scrum）提供了流程级别上的变更响应机制，但是在现代软件开发的信条里，要在代码级别响应变更是最困难的，也是最重要的。本书的宗旨就是竭力为你展示如何编写出能够灵活地自适应变更的代码。

另外，瀑布方法论是以文档为核心的，会产出大量的文档，而这些文档并不能直接改善软件产品。敏捷则恰恰相反，它认为能工作的软件就是这个软件产品最重要的文档。毕竟真正的软件行为是由软件源代码定义的，而不是由与代码相关的文档决定的。此外，瀑布方法论的文档和代码是完全分开的，所以很容易就会出现文档没有与代码完全同步的情况。

Scrum设置了一些度量标准，用于反映项目进度和整体健康状况，这些标准与产品的说明性文档是不同的。总体而言，敏捷倾向于有适量的文档以避免规避责任的现象，但是它也不强制要求必须撰写这些文档。如果这些文档不是撰写一次就再也不会被查阅的话，那么有些代码肯定能从这些支持文档中获益。出于这个原因，易用的在线文档（比如Wiki）就成为了Scrum团队很常用的工具。

本章的剩余部分将会更深入地讨论Scrum中最重要的方面，其中讨论的不完全是Scrum，也有与Scrum相关的常见变体。Scrum流程的目标不仅仅是通过迭代的方式改进软件产品，而且也

包括改进开发流程。在Scrum团队特有的状况和上下文中，Scrum流程鼓励团队持续微调来保证整个流程对所有成员来说是合适的。

本章在讨论过Scrum的基本构成后，还会指出它的一些缺陷。这一章是本书的基础，由于Scrum流程承诺拥抱变更,后续章节就会在此基础上详细讲解如何编写出能够自适应变更的代码。一个声称自己可以优雅地处理变更但很难在代码层次实现的流程是没有意义的。

Scrum的不同形态

任何时候，当一个开发团队声明他们遵循了Scrum方法时，通常是说他们自己遵循了Scrum的某种变体（variant）。纯正的Scrum并不包含很多常见的实践活动，它们来源于诸如极限编程（eXtreme Programming，XP）等其他的一些敏捷方法。Scrum有三个分支，它们的实现都逐渐偏离了纯粹的Scrum理论。

增强的Scrum

Scrum并不包含一些常见的实践活动，比如测试先行以及结对编程。但是对于很多团队而言，这些实践活动是对Scrum流程本身很好的补充，因此它们被认为是有辅助作用的实践。当团队从其他敏捷方法（比如极限编程或者看板）引入一些实践活动时候，实际操作的流程应该被称为增强的Scrum。这意味着，Scrum和一些优秀实践活动的组合起到增强而非削弱标准Scrum流程的作用。

弱化的Scrum

一些开发团队声称他们正在实践Scrum，但是却忽略了一些关键点。他们在产品积压工作上按照优先级顺序排列了工作项，在Sprint冲刺中选择了要完成的工作项，而且有追溯会议以及每日站立会议。但是，他们并没有按照故事点估算故事，而是采用真正的时间估算。这种Scrum流程的变体被称为弱化的Scrum。这种情况下，团队尽管在很多方面应用Scrum定义的实践活动，但是实际上却忽视了几个关键的活动。

根本不是Scrum

如果一个开发团队的实践严重偏离了Scrum方法，他们实际上就不是在使用Scrum。这一定会在开发过程中引起问题，特别是当团队成员希望能应用一种敏捷方法而实际的流程却一点都不像Scrum流程时。我发现每日站立会议是最容易被开发团队采纳的Scrum实践活动，但是让团队成员在心中对变更进行相对估算并且积极拥抱变更却很难。当很多Scrum流程的实践活动被团队放弃后，实际运行的流程就根本不是Scrum了。

1.2　角色和职责

Scrum仅仅是个流程，我要反复强调的是，只有团队成员都遵循流程它才会起作用。不同角色的人有着不同的职责，不同的职责则需要采取不同的行动。

1.2.1　产品负责人

产品负责人（Product Owner，PO）的角色在Scrum中非常重要，因为产品负责人是Scrum团队和客户之间唯一的联系。产品负责人要对最终产品负责，他们负有以下相应的责任。

- ❑ 决定要构建哪些特性。
- ❑ 根据业务价值设定特性的优先级。
- ❑ 接受或者拒绝“已完成”工作。

作为项目成功的关键干系人，产品负责人必须时刻准备着为团队服务，给他们清晰呈现项目的愿景。所有开发团队成员都应该清楚地知道项目的长期目标，而且都能够及时清楚地了解到变更的详细信息。在制定短期冲刺计划时，产品负责人应首先安排好要开发什么以及什么时候开始。产品负责人按照软件的发布计划来选择特性，并且在产品积压工作中对这些特性设定优先级。

尽管产品负责人是个关键角色，但是这并不意味着产品负责人在整个Scrum流程中的影响不受约束。产品负责人不能决定开发团队一个冲刺内需要完成的工作量，因为这是由开发团队自身的开发速度决定的。同样，产品负责人不能决定如何实现工作项，因为开发团队会从技术层面上决定一个故事的详细实现方案。当然，在冲刺的过程中，产品负责人不能改动冲刺目标，改变验收标准，或者增删故事。经过冲刺计划会议，目标和承诺的故事已经选定，此时处于运行状态的冲刺就是不可更改的了。任何改动都必须等到下一个冲刺，除非明确中止当前冲刺或者整个项目，然后重新开始。冲刺的这种不可破坏性保证了开发团队能够在整个冲刺进行过程中心无旁骛地朝着既定的目标努力。

整个冲刺期间，无论故事在进行中还是已经完成，产品负责人都要经常去试用进行中的产品，看看特性的状态如何，或者给正在进行的任务提提意见。产品负责人多花点时间与开发团队保持紧密的沟通是很重要的，这样才可以及时应对那些突然出现的意外和混乱。到冲刺结束时，产品负责人不能简单地接受那些声明“已经完成”但却偏离了初始目标的故事，而是应该通过制定好的验收标准来检验一个故事是否已经完成以及是否可以展示。

1.2.2　Scrum 主管

Scrum主管（Scrum Master，SM）负责在冲刺进行过程中为团队隔离所有外部影响，并且处理团队成员在每日站立会议上提到的各种影响开发的障碍。这样才可以保证在冲刺期间，整个开发团队能够高效地朝着当前的既定目标努力。

产品负责人要对做出怎样的产品负责，Scrum主管则要对如何完成产品的流程负责。因此，Scrum主管的职责就是确保整个团队按照既定的流程来完成既定的产品目标。Scrum主管能够主导对流程的改进（比如把冲刺周期从四周缩短到两周），但是他们的权限也是有限的。比如Scrum主管只可以指导开发团队按照Scrum流程开发，而不可以越俎代庖地指定开发团队如何实现一个故事。

作为流程负责人，Scrum主管要负责组织每日站立会议，他们要确保所有开发团队成员都出席会议，并且记录会议纪要以防遗漏某些行动项。不过，这并不代表着团队成员要在会议上给

Scrum主管做工作汇报，每个成员参加站立会议的真正意图是为了让所有与会者都能大概了解到自己的工作项进度和状况。

1.2.3　开发团队

理想情况下，一个敏捷团队由一些全科专家（generalizing specialist）组成。也就是说，团队的每个成员应该能够熟练使用多个领域的技术，而且精通或特别擅长其中的某个领域。比如说，在一个由四名开发人员组成的敏捷团队中，每个开发人员都能胜任所有与ASP.NET、MVC、Windows Workflow以及Windows Communication Foundation（WCF）相关的工作项；但是其中两个开发人员特别擅长Windows Forms，另外两人则喜欢使用Windows Presentation Foundation（WPF）和Microsoft SQL Server。

团队中开发人员的技能重合能够防止团队中出现筒仓现象。筒仓现象的表现是，团队中的各种专家（比如网页开发专家、数据库专家或WPF专家等）各自掌握了产品开发必需的一种知识。Scrum中，代码是由开发团队集体持有的，而这种筒仓现象会妨碍全体开发人员的参与度。因此在组建团队的时候，应该尽可能避免出现这种筒仓现象。此外，筒仓也不利于业务的开展，因为业务的某个领域会过度依赖单个“专家”开发人员的能力，而且这些“专家”开发人员顶着“他们是唯一能完成这些工作项的人”的大帽子，压力也会非常大。

软件测试人员负责保证所开发软件的质量。在开始实现一个故事前，测试人员可能需要规划自动化测试以保证故事的实现符合各种预先定义的验收标准。他们要么与开发人员一起制定计划，要么单独完成。当开发人员完成一个故事时，就会提交故事的实现以供测试，然后测试分析人员会核实软件产品是否按照要求的那样正常工作。

1.2.4　“猪”和“鸡”

Scrum流程中的所有角色都可以划分为两类：“猪”和“鸡”。这种形象的比喻来源于下面的故事。有一天，鸡对它的好朋友猪说：“猪哎，我有个很棒的想法，我们应该开个饭店！”猪也觉得主意不错，很兴奋地问鸡：“那我们给它起个啥名呢？”鸡说：“要不就叫‘火腿和鸡蛋’吧！”猪想了一下就发火了，说道：“门都没有，你只是下下蛋，我却要割肉！”

这个寓言故事很好笑，用在这里只是用来强调一个项目中的不同角色有着不同的参与度。“猪”需要全身心投入到项目中，并且要对它们的产出负责；而“鸡”只是有所贡献，不会深入参与到项目中。产品负责人、Scrum主管和开发团队扮演的都是“猪”的角色，因为他们都需要全身心致力于产品的交付。通常，客户不需要深入产品开发中，因此他们扮演的是“鸡”的角色。类似地，因为执行管理层的支持工作是针对项目本身的，而不是产品，所以也应该是“鸡”而不是“猪”。

1.3　工件

在所有软件项目的生命周期内都会创建、评审和分解细化很多的文档、图表和度量标准。从这个角度看，一个Scrum项目也会有同样的工件。然而，Scrum文档与其他类型项目管理的文档有着不同的目的和类型。所有敏捷流程和严苛流程之间的一个关键区别，就在于文档的重要性上。比如，结构化系统分析和设计方法（Structured Systems Analysis and Design Methodology，SSADM）就特别强调需要撰写很多的文档。这也被笑称为“大设计优先”（Big Design Up Front，BDUF）理论，这种理论相信，如果能够花足够多的时间和人力资源来撰写文档，所有的恐惧、不确定性和怀疑就可以从项目中清除。敏捷流程则致力于减少文档的数量，只保留那些对于项目成功至关重要的文档。此外，敏捷流程更看重代码而非文档，因为代码本身作为最具有权威性的文档，是随时可以部署、运行和使用的。敏捷流程还倾向于所有干系人都直接沟通，而不是写一些很少有人看的文档。总而言之，文档对于敏捷项目依然是重要的，但是它不能取代可工作的软件和沟通本身。

1.3.1　Scrum 面板

Scrum项目日常工作的中心区域是Scrum面板。应该为面板保留几面墙的空间，太小的面板将没有空间展示重要的细节。也许专门为此在办公室建足够大的墙面代价会比较大，但你也可以采用其他的一些性价比较高的办法。比如可以重新利用那些经常被人忽视的、比较大的白板作为Scrum面板，再加上磁铁和金属填充格，就更像一个Scrum面板了。如果办公室是租来的，或者因某种原因不能损坏墙壁，还可以使用“神奇的”白板，即白纸，因为它方便简洁且无需擦除内容。一定要尝试在办公室找一块合适的地方放置Scrum面板，无论选择了什么样的面板，也无论如何放置它，如果在使用了几个冲刺后，感觉不合适，可以随时改变它。对于Scrum流程而言，Scrum面板实物是必备的，其他的工具（比如数字化面板等）永远无法给你那种站在实际面板前的体验和感觉。尽管数字化面板也有它们的用途，但我仍然相信它们只是辅助工具。图1-3展示了一个典型的Scrum面板。

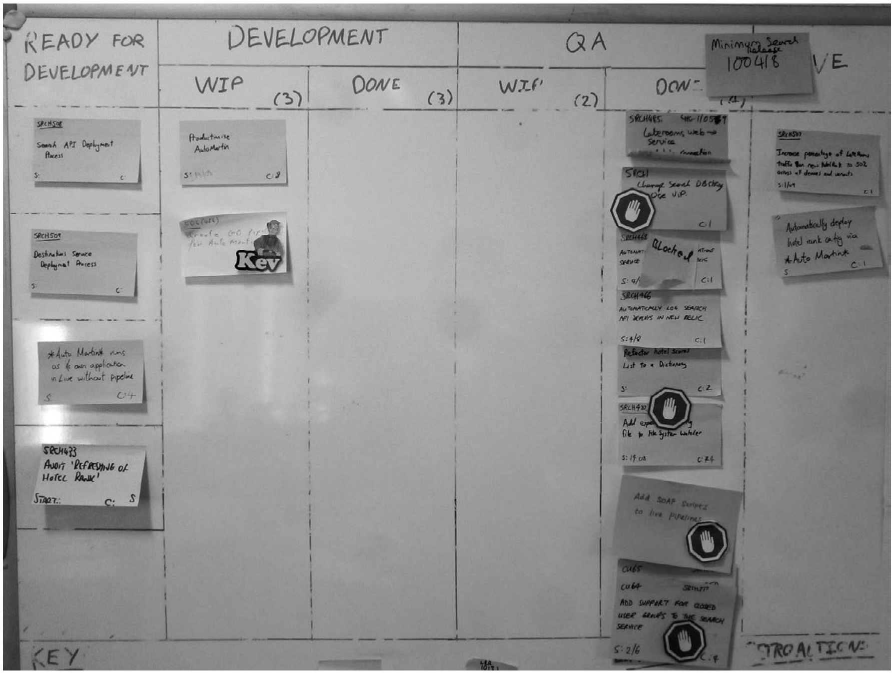


图1-3　一个Scrum面板，展示了当前的开发状态

Scrum面板是项目信息的聚集地，包括了很多细节，并能展现出正在进行的工作项的重要性。下面几节会全面详细地介绍Scrum面板的使用。

1. 卡片

Scrum面板上的主要物品就是卡片。卡片会用来展示软件产品进度的不同元素，从软件的发布到最细小的独立任务。为了清晰起见，不同类型的卡片应该有不同的颜色。因为空间限制，Scrum面板通常只用来展示与当前冲刺相关的故事、任务、缺陷以及技术债务（technical debt）。

> **提示**　单独使用颜色来区分可能无法满足每位团队成员的需求。比如，对于那些无法分辨颜色的团队成员而言，可以辅助使用不同的形状来作卡片类型的区分。

● 组成的层次结构

图1-4展示了Scrum面板上不同类型卡片的层次关系。请注意，这里的前提是产品是由很多个任务组成的。即使是最复杂的软件也可以分解成为有限的若干个独立任务，要完成整个软件，就必须先完成这些任务。

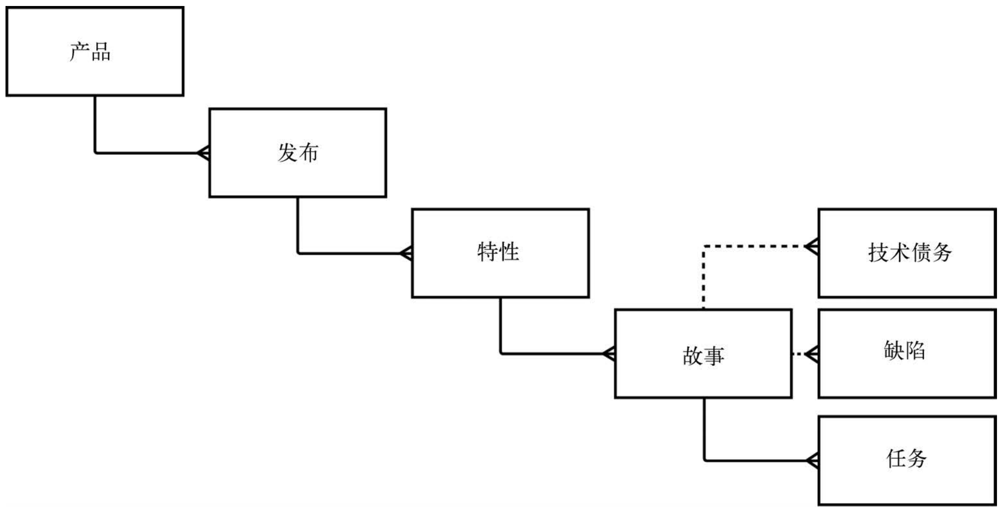


图1-4　Scrum面板上的不同类型的卡片展示了组成产品的不同元素以及它们之间的层次关系

- 产品

Scrum食物链的顶端是要构建的软件产品。软件产品的例子很多：集成开发环境（IDE）、网络应用程序、会计软件、社交媒体应用程序等。你要开发的是软件，要交付的是产品。

通常每个开发团队每次只开发一个产品，但是有时候单个团队也可能会同时为交付多个产品负责。

- 发布

每个要开发的产品都会有多个发布。一个发布就是一个特定版本的软件，用户可以购买软件或者使用该软件提供的服务。有的发布只是为了解决若干缺陷，有的发布则会为关键用户提供有价值的新特性，还有的发布只是提供测试版本来让用户尝鲜和反馈。

网络应用程序通常只会在所有发布前部署一次，版本变更也不会很明显。实际上，Google Chrome网络浏览器就是个很有意思的范例。尽管它是一个桌面应用，但是它的每次发布都很小，对用户而言几乎没有感觉，这与其他浏览器的高调发布方式截然相反。像Internet Explorer 8、9和10分别都有自己的广告投放，而Chrome则没有选择这样的发布模式，Google只是简单地为浏览器本身做推广，推广的时候也不会提及版本。而这种迭代式的发布方式也变得越来越流行。Scrum在每个冲刺后能够很自然地通过专注于交付可工作的软件来支持这种发布模式。

最小可行发布

第一个发布可以是一个**最小可行发布**（Minimum Viable Release，MVR），它包括了能够满足用户最基本需求的一组特性。比如，对于会计软件，最基本的特性集合包括：创建新客户，客户账户上的存取交易以及账户金额汇总。最小可行发布的核心目的就是引导项目尽早

做到自筹资金并继续发展。尽管最小可行发布的结果不太可能就达到自筹资金这个目标，但是至少通过这种发布可以提前获得一些回报来抵消部分开发成本。此外，即使最小可行发布的目标客户范围受限，它的部署也能够获得宝贵的用户反馈，或许还会影响到后续该软件发展的方向。这种及时调整方向的能力也是Scrum（以及其他敏捷方法）自身与生俱来的，因为这些敏捷方法都认为软件不断进化的前提就是软件必须要变化。

无论发布的目的和方式（或者频率）如何，理想情况下，单个产品的生命周期内都会包含多个发布。

- 特性

每个发布由上一个发布的软件没有的若干个新特性组成。任何软件的版本1.0和版本2.0的最显著的区别就是那些团队成员认为能够吸引新老用户购买软件或者升级包的新特性。

最小可售特性（Minimum Marketable Feature，MMF）这个术语可以用来界定一个发布需要的特性集合。下面列出了一些示例特性，这些特性非常通用，可以应用于许多不同的项目，同时也非常特定，它们就是真实世界的特性。

- 以可移植的XML格式导出数据
- 需要在0.5秒内响应网页请求
- 保存数据以备后续使用
- 复制和粘贴文本
- 与同事在网络上共享文件

特性如果能给客户带来价值，那么它就是可以出售的。在尽量提炼出最小功能集合的同时依然能保持其具有可交付的价值，此时得到的特性的粒度也是最小的。

史诗/特性、最小可售特性与主题

当谈论Scrum的时候，可能更经常使用的术语是**史诗**（epic）而不是**特性**（feature），我个人不太经常使用前者。史诗和特性通常被看作“大型的故事”，也就是说，这些故事比最小可售特性大很多，因此无法在一个单独的冲刺后交付。

特性也与Scrum的另外一个术语**主题**（theme）类似，主题是指一组有着共同目标的故事。

对每个发布而言，可以将特性划分为三类：必需的、可选的和想要的。这三个分类是互斥的，用于反映每个特性的优先级。通常，开发团队必须首先完成所有必需的特性，然后才是可选的特性，此外，只有在时间允许的情况下才可以触及想要的特性。你也许已经猜到，这些分类和特性本身总是可变的。当团队想要切换工作焦点的时候（当然，这会附带引起计划和经费的改变），可以在任意时间中止它们、对其重新排序、交换其顺序以及废弃它们。Scrum中的一切都是不确定的，而本书就是致力于帮助你学会应对这些不确定性。

● 用户故事

用户故事可能是绝大多数开发人员最熟悉的Scrum工件，但实际上它根本不是由Scrum定义和提出的。用户故事最先是极限编程方法定义的工件，因为它在软件开发领域变得很常用，因此Scrum方法后来也引入了它。用户故事需要通过下面的模板来声明。

“作为［某个用户角色］，我想要［做某种行为］，以便于［给这个用户角色带来某种价值］。”

上面模板中，方括号中的内容会因具体用户故事而有所不同。下面举个具体的例子来作进一步的解释。

“作为一个注册但未经验证的用户，我想要重置我的密码，以便于当我忘记密码的时候还能够再次登录系统。”

这个用户故事描述示例包含了很多值得注意的地方。首先，这个故事并不包含足够的信息来实现必需的行为。特别要注意的是，用户故事是从用户的角度出发编写的。尽管这一点看起来很明显，但实际上很多人都忽略了这一点，很多故事都是错误地从开发人员的角度编写的。这就是为什么用户故事模板中的第一部分“作为［某个用户角色］”至关重要的原因。类似地，第三部分“以便于［给这个用户角色带来某种价值］”也同等重要，因为如果没有它，那么这个故事存在的根本原因就会被忽略掉。通常这部分也表达了用户故事与父特性之间的联系；上面这个用户故事示例可能属于诸如“被忘记的凭据是可恢复的”这样的特性。也很可能与“用户忘记了登录名”以及“用户忘记了登录名和密码”这两个用户故事归属于同一主题。

既然并不能单单从用户故事开始开发工作，那么它的价值何在？用户故事代表了开发团队与客户之间的对话。当需要实现某个用户故事时，开发人员会带着故事与客户讨论具体的需求，并且会产生出在整个用户故事的生命周期内都必需遵守的若干个验收标准，仅可以将完全符合了所有验收标准的用户故事标注为已完成。

当需求收集完之后，开发人员就需要开始准备一些设计来满足这些需求。这个阶段可能需要使用Balsamiq、Microsoft Visio或者其他一些工具来制作用户界面的模型。通常使用统一建模语言（Unified Modeling Language，UML）图表来从技术角度上详细表达如何改动现有代码以满足这些新的需求。

当设计确定后，团队就可以开始将用户故事分解为小的任务，然后通过完成这些任务来实现整个故事。当开发人员认为故事的实现已经满足要求了，那么他们就可以交付故事实现以进行验收测试。最后的质量保证（quality assurance，QA）阶段会再次根据验收标准来对可工作的软件进行测试验收，并决定批准或者驳回该故事的实现。如果用户故事的实现得到了批准，那么才真正完成了这个用户故事。

让我们花点时间来回顾一下前面讲述的要点。以一个用户故事为依据，开发人员在分析阶段收集需求，然后准备好设计方案，接着做具体实现，最后按照验收标准做测试。这听起来就像是一个瀑布开发方法的过程。这的确是用户故事的要点，为每个用户故事执行一遍小规模的但完整的软件开发生命周期。这种方式有助于防止出现无用功，因为在没有分解和实现用户故事之前，

开发人员不需要做什么，但是他们一直知道用户故事依旧符合软件产品的价值主题。

用户故事是Scrum过程中最主要的工作重心；Scrum流程是通过使用用户故事的故事点数来激励团队成员的。在冲刺计划会议上，团队一起给每个故事估算故事点数，当团队完成了某个用户故事时，就认为团队已经挣取了它的故事点数，并可以从整个冲刺的总点数中扣除该故事的所有点数了。后面的章节会进一步详细讲解故事点数的概念和应用。

- 任务

任务是比用户故事还要小的工作项。可以把一个用户故事分解成多个容易管理的任务，然后分配给多个开发人员并行开发。我倾向于在准备实现故事时再开始做任务分解，但是也有很多人会在冲刺的计划会议时就完成了所有承诺要完成的故事的任务分解。

尽管用户故事是在功能角度上的完整竖切，但是分解得到的任务依然可以分层，这样就可以充分利用团队成员的技术特长。比如说，要给一个数据表增加一个新的数据项，这很可能需要改动用户界面、业务逻辑以及数据访问层实现。团队可以将这个故事分解成三个任务，它们分别针对分解得到的三层需求并指派给三个各有特长的开发人员：WPF高手、C#能手以及数据库大咖。当然，如果团队成员全都是全能技术能手，任何人都可以在任何时间胜任任何任务，那么你就太幸运了。这种分层分解任务的方式可以让每个团队成员拥有很广的选择任务范围，也有助于他们理解整个项目并有更高的满意度。

竖　切

在我小的时候，爸爸每年圣诞节都会做蛋奶层叠蛋糕。这是一种传统的多层英国甜点。最下面是水果块层，然后是松糕层、果酱层和牛奶沙司层，最上面是奶油层。我哥哥喜欢用勺子一直从上朝下吃，而我则喜欢换着吃不同的层。

好的软件设计就像层叠蛋糕一样分层。最下面是数据访问层，然后是对象关系映射层、域模型层、服务层和控制层，最上面是用户界面层。与吃层叠蛋糕一样，开发这种分层软件也有两种方式：横切和竖切。

如果选择横切方式，每层都需要作为一个整体来实现。但是这样不能保证每层的实现都协调同步。用户界面也许已经可以允许用户去交互完成某些特性，但是下面的功能层还没有实现。实际的结果就是用户必须要等待所有层都配合完成这个功能后才可以使用这个应用。这会导致无法及时得到用户的反馈，并且会增加做无用功或者做错的可能性。

在敏捷流程里，应该选择竖切方式。每个用户故事应该由每层上的功能组合而成，并且与最上层的用户界面联系起来。这样才可以完整演示某个功能来获取用户的及时反馈。这种方式也可以避免从程序员角度撰写用户故事，比如“我想能查询数据库以知道那些本月没有缴费的用户清单”这种描述听起来像个任务；而正确的故事描述应该是“生成未付费账户的报告清单”。

请注意，只有用户故事才持有故事点数，用户故事分解得到的任务并不能继承和持有故事点数。可以说五个故事点数的用户故事被分解为三个独立的任务，但是不可以说两个一点的任务和一个三点的任务组成了五个点数的用户故事。这是因为完成单独的任务并不会获得故事集点数的

激励。只有当测试人员和产品负责人在冲刺结束前确认整个用户故事已经完成了才可以获取该故事的所有用户点数。如果没有完成或只是部分完成，整个用户故事的故事点数依然不能从冲刺故事点数中扣除。理想情况下，用户故事会在一个冲刺内完成，如果没有完成，应该在下一个冲刺继续。如果一个故事因耗时太长而无法在一个单独的冲刺内完成，那说明这个用户故事的粒度太大，应该分解为多个更小、更容易管理的故事。

● 技术债务

技术债务是个很有意思的概念，但是它也很容易被误解。它是个隐喻，用于描述在一个用户故事生命周期内在架构设计和实现上所做的折中和妥协。本章后面专门有一节讲解技术债务。

● 缺陷

如果某些完成的用户故事没有符合某些验收标准，就需要创建缺陷卡片了。这就要求有自动化的验收测试：针对某个用户故事撰写的一组测试就形成了一个回归测试套件，可以用于保证后续的工作不会破坏已通过验收的用户故事的实现。

与技术债务一样，缺陷卡片也不会持有故事点数，因此修复缺陷和技术债务带来的问题并不会获得故事点数激励。虽然做到零缺陷和零债务很不现实，但是开发人员应该尽最大努力去避免引入缺陷和技术债务。

所有软件都有缺陷，这是软件开发无法逃避的事实，缺乏计划或不勤勉并不是人会犯错误的根本原因。缺陷通常会被划分为三大类：灾难缺陷（Apocalyptic defect）、行为错误（Behavioral error）和外观上的问题（Cosmetic issue）；按照首字母也被称为A、B、C三类。

灾难缺陷会直接导致应用程序的彻底崩溃或者导致用户的操作无法继续。未捕获异常是一个典型的例子，因为此时应用必须先强制结束然后再次启动，或者是必须重新加载一次网络场景下的网页。这些缺陷应该具有最高优先级，并且必须在软件发布前修复好。

行为错误不像灾难缺陷那样严重但是也会令用户不满。这种类型的错误还有可能比直接让应用程序崩溃更有破坏性。试想一下，如果错误的货币转换逻辑把账户资金数额的小数部分给弄没了，无论算法的错误是对用户还是业务有害，有人终归是要为这样的行为错误承担资金损失的。当然，不是所有的逻辑错误都会这么严重，但是这个例子有助于让我们明白，行为错误可以是中优先级，也可以是高优先级。

界面问题通常是和用户界面相关的问题，比如图片未对齐，窗口无法全屏，或者网页上某个图片无法加载显示等。这些问题不影响软件的使用，只是影响它的外观。尽管这种问题通常只是低优先级的，但是它们也很重要，因为界面的表现也是用户对软件的期望之一。如果用户界面上的按钮无法使用，图片无法加载显示，用户也自然无法相信软件的内部行为会正常工作。相反地，一个花里胡哨的用户界面也会让用户感到这个软件并不是针对公众发布的。信誉不好的项目通常需要重新设计用户界面（甚至可能改变产品品牌）来改善市场对该产品的看法以及重设用户对该产品的期望。

让卡片意图更明确

有很多种办法可以自定义以及个性化Scrum面板上的卡片。

颜色主题

在卡片上应用任何颜色主题都可以，不过按照我的经验，其中一些比较合理。特性和用户故事可以使用索引卡（index card），任务、缺陷和技术债务可以使用即时贴（sticky note），因为可以很方便地把它们贴在相关故事附近。下面是我推荐的使用方式。

- 特性：绿色的索引卡
- 用户故事：白色的索引卡
- 任务：黄色的即时贴
- 缺陷：红色或粉色的即时贴
- 技术债务：紫色或蓝色的即时贴

注意，作为使用卡片最多的用户故事和任务，它们应该使用最常见可用的索引卡和即时贴。此外为了避免索引卡不够用，请尽量使用最常见且可用的颜色。

谁来创建卡片？

这个问题的答案很简单直接，那就是“任何人”。当然，在实际应用中也会有些条件。虽然任何人都可以创建一个卡片，但是该卡片的有效性、优先级、严重程度以及其他一些状态值并不能只由创建者单独决定。所有的特性和用户故事卡片必须由产品负责人最终核实，而任务、缺陷和技术债务卡片则只应该由开发团队来核实。

头像

类似于网上论坛、博客和Twitter上的用户头像，团队的每个成员也要有个自己的小画像。让团队成员自己选择自己的头像，这也是Scrum流程中比较有趣的一个过程。当然，需要引导团队成员不要选择有冒犯性的头像，只要保证最终每个人的头像是有区别的就可以。

在迭代过程中，这些头像会被移动好多次，通常是每天移动一次。因为Scrum面板上已经贴满了故事索引卡和任务即时贴，因此这些头像不应该大于两寸照片。使用薄板覆压图像，可以防止其出现折角和破损，还可以用小片胶带把头像固定在一个地方。

2. 泳道

在Scrum面板上，通过多个垂直线划分出了多条泳道。每条泳道可以包含多个用户故事卡以表示相关故事在其开发生命周期内的进度。典型的排列是从左到右的四条泳道：积压工作、开发、验收和完成。

积压工作泳道中的故事是开发团队已经承诺在当前冲刺要进行开发的，所以它们迟早会被移动到开发泳道中，除非它们在开发前被中止了。这条泳道内的故事卡可以按照优先级排列，这样最上面的卡片总是包含下一个要实现的故事。

从积压工作泳道中取出故事后，首先要先和产品负责人沟通以确定该故事的范围和需求，然后分解为多个独立的任务，最后把该故事和分解它得到的多个任务一起放进开发泳道中。此时，所有参与开发该故事的团队成员的小头像也应该贴在故事卡周围了。不同的泳道会有不同的限制。比如，你也许要求最多同步开发三个故事，这样才可以强迫团队优先完成已经开始的故事，而不是再从积压工作泳道中取出新的故事。请切记：没有完全完成的故事无法挣取该故事的任何点数。

当一个故事经过分析、设计和实现后，它会被标记为“已经完成开发”的状态，此时就可以把它移动到验证泳道中了。理想情况下，验证环境应该和产品环境尽可能地一致，以避免后期部署时因为环境的些许不同而发生错误。测试分析人员使用验收标准来评估故事。本质上讲，他们要做的是尽量破坏故事以证明代码无法按照预期正常工作。通常，测试人员通过给某些操作输入边界和错误数据，来验证操作代码的验证逻辑是否工作正常。测试人员甚至会尝试找出安全漏洞，以确保可疑终端用户无法获取额外的高级别权限。测试验证通过之后，所有和该故事相关的故事点数就可以从当前冲刺的总故事点数上扣除了，同时冲刺燃尽图（展示冲刺进度的图表）也可以进行更新了。这些工件将在后面详细讲解。

- 水平泳道

Scrum面板还可以通过水平泳道来做进一步的划分。团队可以使用水平泳道按照特性来给故事分组，这样团队所有人一眼就能看得出当前正在攻关哪里，也能知道需要缓解哪些瓶颈。

在面板顶部有个特殊的泳道称为快速泳道，可以把所有优先级非常高的任务放在这条水平泳道。通过指定多名团队成员集中在快速泳道上的工作项，来尽快完成这些高优先级的任务；但同时，这也会对其他未完成的任务产生不好的影响。集体攻坚可以确保团队停下手上的工作，以集中力量协同解决一个最高优先级的问题或者任务。这个办法对这种出现最高优先级问题的场景很有用，但是也应该慎用。快速泳道中最常见的工作项就是在已经发布的产品中发现的灾难缺陷。

3. 技术债务

技术债务这个概念值得做进一步的解释。在实现一个故事的过程中，很有可能需要在“最优代码”和“足够好的代码”间做出一些折中以确保不错过最后期限。当然这并不意味着为了赶工期，就可以心甘情愿地容忍（不是积极地鼓励）糟糕的设计，但是当前先简单实现，后面再做改进也是很有实际应用价值的。

- 技术债务的好坏

在项目的生命周期内，债务可能会逐渐累积。之所以使用术语债务（debt）是因为这是一个很好的隐喻，用于描述应该如何看待出现的问题。有些类型的资金债务没有一点错，比如，你准备买辆车但无法一次付清车款，正好有机会选择分十二个月无息贷款购买，虽然此时你有负债了，但从长远看，这个选择负债的决定也是好的。这辆车帮助你赚回了每月要支付的分期款，因为你现在可以开车按时上班了。

当然，有些负债并不是好事。比如你在不清楚如何还款的情况下用信用卡买了一些奢侈品，最终肯定会陷入一个总在以最低限度支付利息的恶性循环中。直到事后，你才会发觉这是一个多么糟糕的债务决定。问题的关键点在于，首先要仔细分辨候选项，然后再决定是要背负债务还是要提前全款付清。

软件开发也有同样的折中。你可以选择暂时先实现一个次优的方案来确保不错过最后期限，或者选择花费更长时间来改善设计但是会错过最后期限。选择没有绝对的对错，只有分析具体情况才能判断引入技术债务是对还是错。

- 技术债务象限图

Martin Fowler是一位非常杰出的敏捷先行者。他提出了技术债务象限图的定义，用于对完成

故事可能需要做出的折中和妥协进行归类。象限图的x和y轴将一个平面划分成四个象限，x轴代表问题“是否在为正当的原因积累技术负债？”，y轴代表问题“是否有其他的替代方案可以避免技术负债？”。

如果你对第一个问题回答“是”，那么你在引入一个有着长期利益的技术债务，因为你能指出增加债务的正当理由而且你也很清楚要注意什么。如果你回答“否”，那么这个债务是有负面后果的，你最好现在就处理掉这个债务，而不应该允许这种债务继续增加。

对于第二个问题，肯定的回答代表你已经考虑了债务的替代方案且决定当前先引入这个技术负债。否定的回答则表明了你并没有为引入的技术债务认真考虑过其他的替代方案。

问题的答案组成了图1-5展示的四个可能的场景。

- 不计后果的，有意的：这种类型的负债是最糟糕的。相当于在说“我没有时间设计”，这代表了一种很不健康的工作环境。出现这样的决定时，就应该给每个成员提出警告：现在的团队不能适应当前状况，而且正在走向注定失败的结局。
- 不计后果的，无意的：这种类型的债务大多数是因为缺乏经验才引入的。这是由于开发人员不够了解现代软件工程的最佳实践导致的。很像上一个场景，代码很可能是乱糟糟的，但是开发人员不知道还有更好的可供选择的替代方案。解决方案就是再学习，只要开发人员愿意学习，他们就可以停止引入这种类型的技术债务。
- 谨慎的，无意的：这种类型的债务发生在当你遵循了最佳实践时，却发现仍有更好的方法。“虽然现在没做，但是已经知道以后该怎么做了。”这和前一种场景类似，不同的是，这种情况发生时所有的开发人员的意见是一致的，那就是当时他们都不知道还有更好的方案可供选择。
- 谨慎的，有意的：这是最令人满意的债务类型。你已经认真考虑到了所有的候选项，也很明白自己正在做什么，也知道为什么要引入这个债务。引入这种类型的债务后，通常会伴随一个这样的决定:“现在先发布，然后再处理那些已经考虑清楚的问题”。

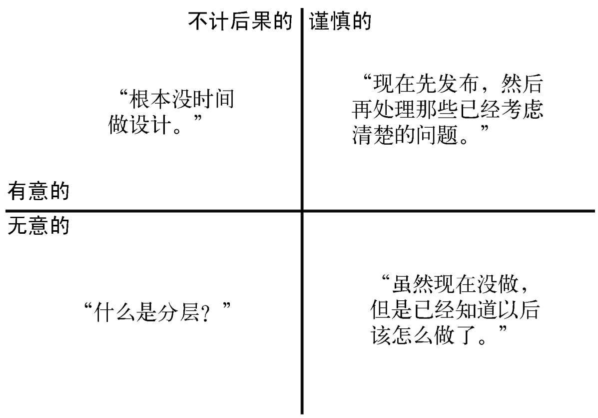


图1-5　正如Martin Fowler所描述的那样，技术债务象限图能让开发人员明白技术债务可以归为四类

- 债务偿还

技术债务和缺陷一样，不能直接持有故事点数，但是即使缺乏直接的激励，作为债务依然是必须要偿还的。最好的方式是给故事再附加一个技术债务卡，并且重构代码以实现新的设计和行为。再从面板中取候选的故事时，就需要检查是否有任何要改动的代码带有技术负债，如果有，就将该技术负债和该故事放在一起处理。

4. 数字Scrum面板

如果不把数字Scrum面板一直挂在墙上，就一定会隐藏很多重要的项目信息。通过开放信息并让整个公司都看到它们，你可以邀请人们对流程提出问题。流程越透明越好，特别是当你在公司第一次引入Scrum实践时。它鼓励获取重要干系人的反馈，你应该很好地引导这些干系人参与到流程中来。

话虽老套，但是人们的确很害怕改变。害怕和焦虑是人本身对未知的自然反应。通过告知人们你在做什么和特定图表表达的意义（以及为什么墙上会有很多的索引卡），你能将协作沟通的积极态度推广给了大家。此外，给一个外行解释这些信息对你也是有好处的，因为你的解释过程或许会让你自身对整个流程有了更清楚的理解。

所有的工具中，最好的总是那些易接触和没有约束的工具。这些工具会被频繁自由地使用，而当一个工具开始变得有些不便使用时，它也逐渐会被抛弃，那些原本被经常使用却未能与时俱进的工具也会迅速地被人遗忘。

5. 完成的定义

所有项目都需要*完成的定义*（Definition of Done，DoD）。为了验收判断是否完成，所有用户故事都必须符合清晰的完成的定义。下面是开发人员经常提及的，你有听到过多少次呢？

> “做完了，不过我还要再做一下测试……”
> “做完了，但是我刚刚又发现了一个问题，可能需要修复……”
> “做完了，只是我觉得设计还不完美，我计划改一下接口……”

我过去也用过这些托辞。如果真的做完了，那就说“做完了”，不需要再附加任何说明、条件或解释。上面这几个例子实际上代表着开发人员还需要多一点时间，因为他们原先的估算不够好或者遇到某个未预料到的问题。所有人必须认同并且至始至终遵守完成的定义。如果一个故事不符合标准，它不能算是真正完成了。只有符合完成的定义的用户故事才可以标记为完成状态。

完成的定义都包含什么？这完全取决于你、你的团队，以及你想要的质量保证流程的严格程度。无论如何，你可以参考下面这个常用的完成的定义。

为了将某个用户故事标记为完成状态，必须确保以下事项。

- ❑ 对所有代码的成功和失败路径都做了完整的单元测试，并且通过了所有测试。
- ❑ 所有代码都无误地提交到了持续集成系统中，编译和构建是成功的，并且通过了所有测试。
- ❑ 通过了验收标准和产品负责人的验收。
- ❑ 不在同一个用户故事下的开发者相互实施了代码评审。
- ❑ 只撰写了适量的用于沟通的文档。

- ❑ 拒绝了那些不计后果的技术债务。

在上面几条的基础上，你可以任意删除、修改或者增加定义条款，但是请务必确保定义是严谨的。如果一个用户故事无法满足所有验收标准，你要么确保这个故事仍然能够满足所有标准，要么把要求比较高的标准从完成的定义中完全删除。比如，如果你觉得交叉代码评审太死板或没必要，那就不要把它包含在你的完成的定义中。

1.3.2 图表和度量标准

在Scrum项目中，有多个图表可以用来监测项目进度。它们能够表现项目的健康状态和进展历史，还可以预测后期可能取得的成果。这些图表要在Scrum面板上有着醒目的位置，尺寸大小要保证较远的时候也能看得清。这种公共的展示方式在向所有团队成员暗示，这些图表不是什么秘密，也不是用来监测项目管理的消耗情况的；相反地，这些图表很直接地展示了项目进度的监测方式。这种公共的展示方式能够告诉所有参与项目的人员，创建这些图表不是为了监控和提高效率，而是用来体现和诊断整个项目中存在的问题。

此外，请尽量避免评测团队中个人层面的任何事情，比如单个开发人员挣取的故事点数。因为这会错误地引导团队个人把自己的进度看得比整个团队和项目的进度更重要。一旦有了这些个人考核标准，开发人员会迅速开始为这些个人目标行动起来，为了不在考核结束时被评为不合格，他们会尽量去独占大型的用户故事以争取这些故事的所有点数。所以，请警惕你对这种个人目标设置的奖惩机制。

> **警告** 请谨慎选择你要监测的事项，因为你的选择会产生“观察者效应”。比如说，对于某些标准而言，在开始监测之前必须先调整测量的对象。再比如，要测量汽车的胎压，就要先给轮胎放点气以改变胎压。人性也存在同样的情况，当团队成员知道他们要被考核时，他们将会竭尽全力去改善他们的数据表现来达到绩效要求。这并不是说团队成员全都是利己主义者，而是说当团队成员认识到故事点数是他们的一个考核标准时，每个人都很可能争相选择那些高性价比的用户故事（也就是说，实际工作量一样但故事点数更多）。这里需要使用三角测量法（1.4.5节中会有讲解）来调整估算工作量和实际工作量的差距。

1. 故事点数

故事点数的意图是激励开发团队为每个冲刺的发布增加商业价值。整个开发团队会在冲刺计划会议（本章后面专门有一节讲解冲刺计划会议）期间讨论并为承诺要完成的用户故事分配故事点数。一个用户故事点数用来衡量实现该用户故事定义行为所需的工作量，其中包括了整个软件生命周期的所有阶段的工作量：需求分析、方案设计、含单元测试的代码实现、测试验收，还有部署实验。尽管每个故事都应当足够小以确保能够在一个冲刺中完成，但是实际故事的规模变化还是很大的。

实现最小的“一个点数的故事”只需要最小的工作量。有个很有意思但也很重要的事实是，

故事点数值只对得出该估算点数值的开发团队才有意义。因为技能和经验值的不同，一个团队的一个点数的故事对另外一个团队来说很有可能变为三个点数的故事。经过多个冲刺后，不同团队对于同一个故事的工作量估算慢慢才会变得接近了。

有一点需要强调的是，故事点数不是用来表达工作量的绝对时长（天数、小时数或者其他时间单位的测量值）。故事点数只是根据以往实际时长范围对工作量进行粗略的估算，如图1-6所示。表中的垂直线表示估算的时间范围，附在垂直线上的水平短粗线表示相应点数的故事实际耗费的时长。

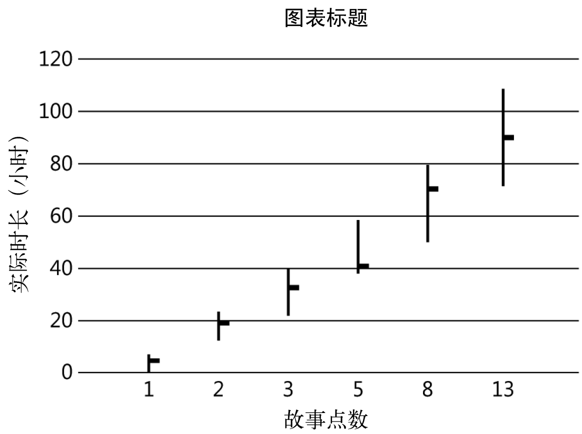


图1-6　最小/最大/平均值表，展示了估算工作量与实际工作量之间的关系

从图1-6中还可以看出：大的用户故事的实际工作量时长范围更大，也应该更难准确预计实际工作量的时长。

2. 速度

经过多个冲刺后，可以开始计算已经完成的用户故事的平均开发速度了。假设一个团队已经完成了三个冲刺，分别完成了8、12和11个故事点数。也就是说，三个冲刺总共完成了31个故事点数，平均每个冲刺完成10个故事点数。“每个冲刺10个故事点数”就是该团队的开发速度，可以按照以下两种方法使用速度值。

第一种方法，团队的速度可以作为团队下一个冲刺承诺要完成的故事点数的上限。如果团队每个冲刺平均能完成10个故事点数，那么在单个冲刺中承诺高于10个故事点数并不只代表太过乐观，还意味着要准备接受更高的失败概率。设置一个可完成的目标，然后完成，甚至超额完成，这总比设置一个不现实的目标然后失败要好。如果团队承诺了10个故事点数，最终实际完成了11，那么新的团队速度就是11：(12+11+11)/3。这就是Scrum流程的反馈机制。

第二种方法，团队可以使用速度分析交付存在的问题。如果团队速度在某个冲刺中明显变慢，这就表明冲刺中有糟糕的事情发生了，需要尽快解决。有可能是故事规模太大并且作出了错误的估算，导致偏差太大，这种大型故事应该需要多个冲刺才可以真正完成；也有可能是因为太多的

关键团队成员同时休假或者生病，从而导致进度变慢；另外一种可能的情况是，团队花费了太多的时间去重构现有的代码，而没有专注在新特性的实现上。无论什么原因，25%以内的速度降幅还不是很严重，但是它已经暗示很可能会出现问题，一旦问题出现，团队应该及时处理。如果速度每周都有下降，而且减速幅度越来越大，这一定表明已经出现了严重的问题，很有可能就是部分代码无法快速适应变化导致的；而本书的主旨就是为了帮助你解决这个问题。

3. 冲刺燃尽图

每个冲刺开始的时候，都要在Scrum面板上创建一个二维平面图，其中，y轴表示总的故事点数，x轴表示工作天数。如图1-7所示，笔直的对角线也称为*最吻合线*（line of best fit），它展示了最理想的冲刺进度中故事点数随着时间扣除的过程。

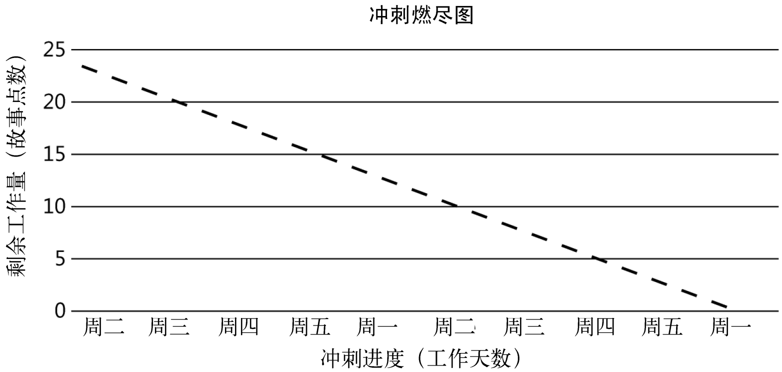


图1-7　一个刚开始的冲刺燃尽图。其中的对角线展示了最吻合目标的冲刺过程（在这个例子中，目标是挣取23个故事点数）

在每日站立会议上，会从总的故事点数中扣除所有从已经完成的故事上挣取的故事点数。图1-8展示了为达到冲刺目标，实际冲刺过程就是一条与最吻合线有偏差的曲线。

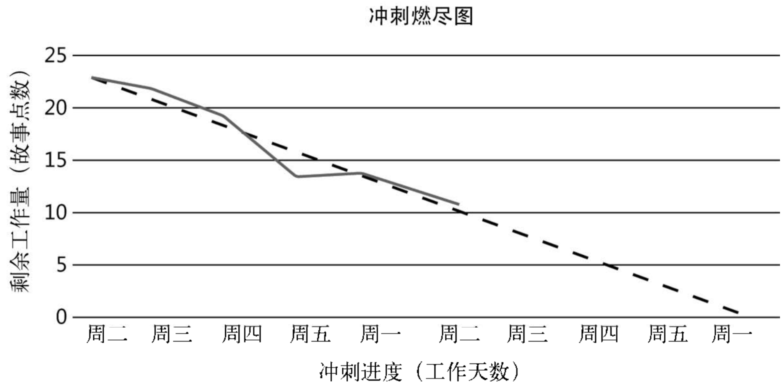


图1-8　一个进行到一半的冲刺燃尽图。在这个例子中，团队正在努力让实际冲刺过程贴合“完美路径”，尽管第一个周五和周一期间并没有任何进展

用不同的颜色绘制实际进度曲线和最吻合线，可以让两者有视觉上的差异。在冲刺过程中，如果实际进度曲线位于最吻合线的上部，这表明有问题出现了，团队能够完成的工作量比计划的要少。相反，如果实际进度曲线位于最吻合线的下方，这表示当前项目进度比计划要快。在整个冲刺过程中，有时会出现实际进度曲线在最吻合线上下微微波动的现象，这是正常的。但是如果实际进度曲线相对于最吻合线的波动过大，那就需要认真查找原因了。

当冲刺需要在固定时间段内完成固定的工作量时，使用燃尽图对项目是很有帮助的。燃尽图中，曲线是无法位于x轴下方的，因为当y等于0时，代表团队已经完成了所有的工作。

4. 特性燃耗图

在一个冲刺中，冲刺燃尽图用于从故事层面追踪实际的进度，而特性燃耗图则是用来追踪特性完成的进度。在每个冲刺结束时，可能会完整实现一个新的特性。特性燃耗图的最大好处就是能防止冒充交付特性，因为图中的曲线根本就没有变化。随着时间的推移，特性燃耗图中的曲线会呈现线性增长，最理想的情况是一直没有出现过平行线。图1-9展示了一个很好的特性燃耗图。

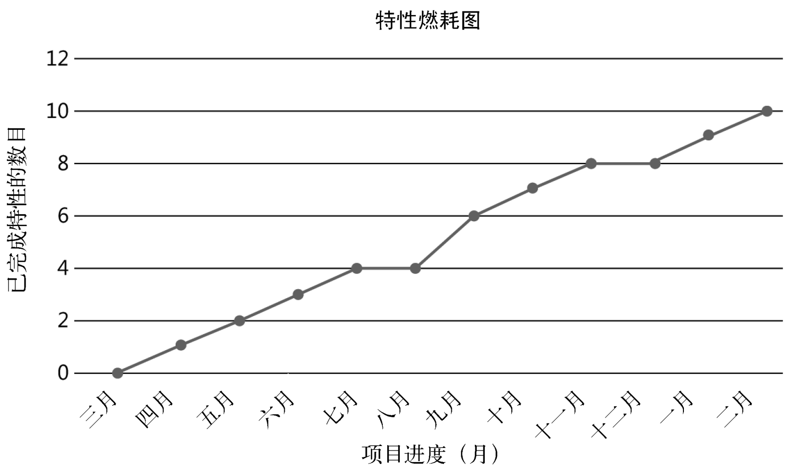


图1-9　这个特性燃耗图展示了一个健康的项目在一年中持续进展的过程

图1-9中曲线坡度可能有些平缓，但它表明了团队的开发节奏很好，正在以一种可预期的速度持续交付特性。虽然与完美的直线有点偏离，但是这很正常，没什么可担心的。

相反，图1-10中的特性燃耗图已经表明出现了开发问题。在开始阶段，团队的状态很好，快速地交付了很多特性，但是后期整整八个月只交付了两个特性，整个团队完全陷入了泥沼。

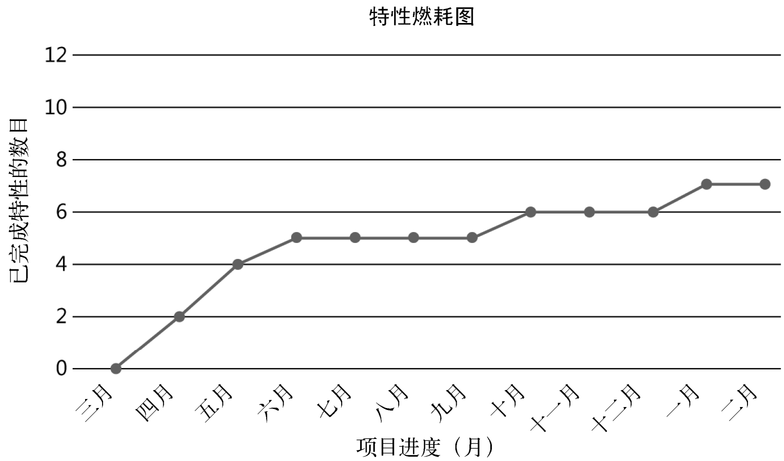


图1-10　这个特性燃耗图展示了一个停滞的项目在一年中缓慢进展的过程

这个问题很清楚：代码无法适应需求的变化。曲线前段的点可能表明团队没有做单元测试，没有做分层架构或者没有采用其他的最佳实践。通过忽视了这些其实很必要的行动，团队设法早早完成了更多的特性。但是，代码逐渐变得愈发臃肿和混乱，开发进度会显著变慢，很长时间才能完整交付一个特性。这种陷于停滞的进度很可能是由于大量累积的缺陷和糟糕的技术债务引起的。如果这个项目继续这种糟糕的状况，最终更好的选择可能是重新开始。长期来看，努力让项目重回正轨的重构工作将会是非常值得的。当然，如果能早早发现问题，团队和项目就能够重新恢复。尽管如此，还是建议从项目一开始就正确地应用各种敏捷开发的优秀实践，而不是等进行中的项目（brownfield project）变得一团糟的时候才引入这些实践。

> **注意**　brownfield是指项目已经处于进行中的状态。与它相反的是greenfield项目，代表新的还未启动的项目。这两个术语都是从建筑产业借鉴而来的。

1.3.3　积压工作

积压工作是一个列表，其中包含了一些需要处理的待定工作项。这些工作项在合适的时间会从积压工作中取出，然后处理直至完成。每个工作项都有自己的优先级和工作量估算值，整个积压工作列表是按照优先级高低和工作量大小进行排序的。

Scrum过程中维护了两个积压工作：产品积压工作和冲刺积压工作，它们分别有着自己特有的用途。

1. 产品积压工作

在产品生命周期内，产品积压工作上总是包含一些待实现的特性。因为产品积压工作中的特

性还没有被取出并放入到冲刺当中，所以开发团队实际上并不会处理产品积压工作上的特性。尽管如此，开发团队（或者团队的代表）仍需要花时间去评估这些特性的工作量。花时间评估特性工作量也有助于对这些产品积压工作上的特性进行排序，以此保证产品积压工作上所有的特性总是排好序的。

每个特性业务价值的高低决定了各自的优先级别的高低。特性的业务价值是由产品积压工作的所有者（即产品负责人）来决定的。产品负责人会向开发团队描述具体业务并为特性定义明确的业务价值。为了挖掘和定义具体特性内在的业务价值，产品负责人必须有丰富的业务知识和工作经验。当产品积压工作上的两个特性的业务价值一样重要时，特性所需工作量的大小会决定二者的优先级。假设有两个高业务价值的特性，相比较其中一个所需工作量要大一些，那么应该首先实现工作量小的特性。因为工作量相对较小的特性，风险会小一些，实现周期也会短一些，相应的投资回报率（Return On Investment，ROI）也会高一些。

当业务需要发布产品的新版本时，有两种方式可以根据产品积压工作为这个发布选择最有价值的特性。第一种是业务需求已经确定了最终的发布日期，需要根据估算的工作量和可用时间从产品积压工作中选择可能完成的特性。第二种是业务需求已经确定了最终的发布特性，只需要根据从产品积压工作中选择的目标特性的估算给出预计的发布日期。

除了特性外，产品积压工作还可以包含那些必须要修复但仍未进入任何冲刺的缺陷。与特性一样，也可以给缺陷指定业务价值。此外，由于缺陷的根本原因还不清楚，因此在确定缺陷修复工作量之前，可能需要时间对它们进行估算。

产品积压工作也应该像其他敏捷文档一样保持开放。所有人都应该能够看到产品积压工作文档，可以在产品开发期间贡献想法，提出建议或指出问题。同样，产品积压工作需要时刻展示权威的、真实的项目状态。很多时候，错误的决定都源自不准确的信息，根据过期的产品积压工作制定的关键发布计划执行起来肯定会是一团糟。

2. 冲刺积压工作

冲刺积压工作包含了所有在即将开始的冲刺中承诺要完成的用户故事。冲刺开始的时候，团队会根据当前开发速度和待开发故事的工作量估算为冲刺选择足量的工作。选定故事后，团队可以开始将所选故事分解为任务，并使用实际时间单位小时来度量这些任务的工作量。最后，团队中每个开发人员再为自己选取足量的任务。

开发团队全权负责冲刺积压工作及其工作量估算。其他人没有权力给冲刺积压工作增加工作项，也不可以为开发团队指定工作量估算。尽管详细的冲刺积压工作是由开发团队单独定义，但是它必须要基于有优先级排序的产品积压工作。

1.4 冲刺

Scrum项目的迭代被称为冲刺。一个冲刺周期最短一周，最长四周，最常见的是两周。如果冲刺周期太短，开发团队没有足够时间来完成既定目标，太长又容易让团队失去工作焦点。

冲刺通常有数字索引，从冲刺0开始。冲刺0的目的是让团队准备开发环境以及在后续真正的

冲刺开始前召开一些主要的计划会议。冲刺0很可能没有故事点数，但是冲刺0中完成的准备工作会对过渡到后续Scrum冲刺有很多好处。

大家很容易就会想按照工作周来安排冲刺，也就是说从周一开始，到周五结束。这种安排的问题是冲刺回顾（后面章节会简短讲解）会需要大量的会议时间，但是没有人喜欢在周五的下午无精打采地坐在会议室开半天的会。因为通常周五很多人都打算早早离开办公室，而且周末马上到了，大家精力都不够集中，工作效率很可能会下降。同样地，也没有人喜欢每周上班的第一天就开半天的会。因此，最好的安排是在周中（周二、周三或者周四）启动冲刺以避免上面提及的问题。

下面按照出现顺序讲解冲刺中出现的所有会议。

1.4.1　发布计划会议

软件的发布计划必须在冲刺开始前就准备好。为此，用户和产品负责人需要在发布计划会议上决定发布日期以及该发布所包含特性的优先级排序和工作量估算。

1. 特性工作量估算

无论特性的规模多大，都可能会有大量的工作要做。因此，任何尝试准确预计所需工作量的效果都有可能大打折扣。鉴于这个原因，特性工作量的估算可以使用常见的T恤码号来计量。

- 特大码（XL）
- 大码（L）
- 中码（M）
- 小码（S）
- 特小码（XS）

2. 特性优先级

特性的优先级排序也很重要，因为有时很难预计一个实际的发布最终能包含多少个特性。在一个特定的发布中，需要给每个特性指定下面三种优先级中的一种。

- 必需的（Required）
- 首选的（Preferred）
- 想要的（Desired）

所有必需的特性构成了最小可行发布。首选的特性是在冲刺时间有盈余的情况下才需要处理。想要的特性优先级是最低的，它们都只是一些锦上添花的特性（当然，用户肯定会想如果能在发布中包含这些特性那也挺好的）。

此外，业务干系人要给所有特性编号，以便开发团队能够按照明确的优先级顺序实现每个特性。

1.4.2　冲刺计划会议

冲刺计划会议的输出应该是所有承诺要在该冲刺中完成的用户故事的估算。与Scrum的其他

流程一样，用户故事估算流程也有很多种变体。本节主要讨论计划扑克和亲和估算，前者是很常见的一种估算讨论的方式，后者是一种快速估算故事相对大小的方法。当用户故事数量比较多的时候，亲和估算会更合适一些。在一个单独的冲刺内，当要估算的用户故事数目不多时，可以使用计划扑克，而当故事数量太大或者时间太短时，就可以使用亲和估算。

1. 计划扑克

计划扑克讨论环节需要整个开发团队参与，包括业务分析人员、开发人员以及测试分析人员，此外还包括Scrum主管以及产品负责人。开始讨论时，首先对产品积压工作上每个用户故事作一些详细的介绍，然后要求每个人用故事点数来给故事的大小投票。

为了避免故事点数都比较接近的情况，最好事先对可选点数进行限制。比如，常见的一组故事点数是经过修改的斐波那契数列：1、2、3、5、8、13、20、40和100。无论选择何种点数序列，总的故事点数的个数必须是有限的，而且点数之间的间隔应该逐渐变大。0作为最小的点数值，可以加到序列中来表达“无需任何工作”，因为有些故事的工作量几乎是可以忽略不计的。序列中最大的点数值用来表达该故事规模太大，无法在一个冲刺内完成，在真正进入开发前需要作进一步的竖切。

投票时可以使用真正的扑克牌，不过，利用闲置的索引卡甚至直接写上数字的白纸也不错。投票时，所有人必须同时展示出自己卡片上的故事点数，这样可以避免在确定故事点数时相互干扰。当然投票结果不是每次都可以达成一致的，很多时候团队成员间会出现较大的分歧。这种现象非常正常，因为总会有个别人出现估算偏差的，他们需要在达成一致的目标下重新考量自己的估算。比如，当平均点数是8而有人却投了1时，会议主持者应该（礼貌地）要求该投票人解释他认为所投故事工作量这么小的原因。同样地，超过平均值过多的投票者也需要给大家解释他们认为工作量很大的原因。整个讨论都是为了得出一个整个团队都认可的、实现该用户故事所需的故事点数。

当投票者讲述了理由之后，可能需要重新投票，因为其他人也许发现自己的估算过大或过小了，而刚才偏离平均值的人才是正确的。最终，所有人会达成一致并得到合适的故事点数。当已经估算的用户故事点数之和达到团队当前的速度时，就可以停止讨论和评估剩余的故事了，因为每个冲刺能够选择的最大工作量应该不大于团队当前的开发速度。

> **避免出现帕金森定律所揭示的现象**
>
> 帕金森定律指出：
>
> **“大多数情况下，人们会应付工作直至耗尽所有可用的时间。”**
>
> 当你不使用实际时间单位来估算用户故事的时候，出现帕金森定律所说现象的几率也就变小了。团队从始至终都应该聚焦在尽快完成故事的行动上，也就是说尽快满足完成的定义。

2. 亲和估算

亲和估算的提出是为了解决计划扑克的局限性，因为如果故事数目很大时，使用计划扑克方

法达成估算需要耗费大量的时间。使用亲和估算方法，团队无需逐个讨论每个故事，只需要从产品积压工作中提取两个优先级最高的故事并对比它们的工作量大小，然后将小故事的卡片放在桌子的左侧，大故事的卡片放在桌子的右侧。

然后，团队成员再从产品积压工作中取出优先级最高的一个故事，并根据该故事的大小把它放置在已有的小或大故事的周边。如果放在小故事左侧，代表更小；放在大故事右侧，代表更大。如果放在某个故事（不论大小）上面，代表与该故事大小基本一致。如果放置于两个故事之间，则代表着大小在二者之间。如此反复，直至觉得当前冲刺的故事工作量已经达到饱和。

这种方法按照相对大小对故事进行分组，团队可以按照计划扑克方法中提及的经过修改的斐波那契数列从左至右地给所有故事组分配故事点数。如果要估算的故事很多或没有时间逐个估算，此时亲和估算就是一个很好的办法，因为它能根据故事的相对大小很快得到所有故事的点数估算。

1.4.3　每日站立会议

尽管有几个Scrum会议会持续好几个小时，但是平时几乎感觉不到Scrum流程在运作，只有在每日Scrum会议上才能看到，这个会议也被称之为“每日站立会议”。

召开这个会议的时候，所有团队成员都要围着Scrum面板站立，每个人都要面对着其余团队成员发言。每日站立会议的持续时间最长不应超过十五分钟。为了保持会议的高效，每个人都应该回答下面三个问题。

- 昨天做了什么？
- 今天计划做什么？
- 遇到了什么障碍？

每日站立会议的重点是给所有团队成员公开昨天的实际进度以及今天的预计进度。在讲述昨天完成了什么的时候，你可以自由更新Scrum面板，把卡片从一个泳道挪到另外一个泳道，或把小头像从一个卡片挪到另外一个卡片上，同时概要地讲述昨天的工作情况和感受。如果当前手头上没有任务了，你需要在会议上告知Scrum主管并申请新的工作项。障碍是指所有可能影响你达成今天目标的事情。因为你需要在明天的站立会议上声明你今天做的事情，所以记录下任何有可能影响你完成计划的事情非常重要。障碍有可能与工作内容直接相关，比如“如果网络还像昨天那样断开，我将无法继续今天的工作”，或者只是与工作无关的个人事宜，比如“我下午两点预约去看牙医，所以我很可能无法完成计划”。不论障碍是什么，Scrum主管都应该记录下来以了解所有人手头上的故事进度和状态。

表情日历（niko-niko calendar）

表情日历有时也称为“情绪面板”（日语中，niko-niko的意思是“笑脸”），它作为一个晴雨表能很好地反应团队成员在冲刺期间对进度的感受。表情日历画在Scrum面板上，行名是冲刺的工作日，列名是成员名字，如图1-11所示。

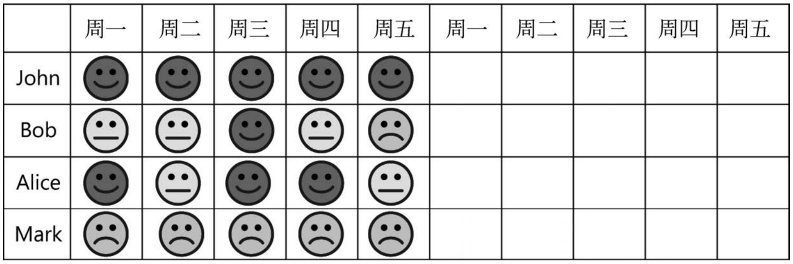


图1-11 表情日历可以清楚明了地展现出了成员冲刺状态的好坏

在每日站立会议上，每个人都要在自己昨天的日历格子上放置一个贴纸。这种贴纸有绿、黄、红三种颜色，分别代表昨天的工作状态：好、一般、糟糕。表情日历能清楚明了地展示出，有些成员因为进度一直不好需要帮助，只是他们可能不想主动提出来。

表情日历只是另外一个有助于促进反馈效率的测量方法。如果所有团队成员持续地对他们手头的工作提不起精神，此时可能需要提振一下团队的士气了。如果只有团队中某个成员一直感觉很糟糕，而其他人状态都还不错，这说明这个成员进度落后太多或者根本不喜欢自己手头的任务。最糟糕的情况是，冲刺快要结束时整个团队还自我感觉良好，但实际上代码一团糟，客户都在敲门要账了，这只能说此时的团队已经根本不在乎项目成败了。

每个人发完言后，站立会议就结束了。Scrum主管要特别留意的是，不要让大家的发言跑题，因为大家会非常容易陷入问题的讨论当中。如果有人提及在代码中看到了有着奇怪有趣表现的问题时，所有开发人员会立即不由自主地开始猜测可能的原因了。要清楚会议中有其他与会者，难道测试分析人员真的需要旁听有关Microsoft Visual Studio几乎使用了机器的所有可用内存的讨论吗？当然不需要。正确的做法是，Scrum主管先记录问题，会后再组织相关的人一起开展讨论。

1.4.4 冲刺演示会议

在冲刺日历上，冲刺演示是一个关键的会议。这个会议要在真实的产品环境下演示所有已经完成的故事，也就是说那些符合完成定义的故事。开发团队所有人员必须到场，也可以邀请其他干系人，比如管理层和销售团队的代表。此外，任何对项目进展感兴趣的人都可以自由观摩演示。所有项目都应该持有这种积极开放的态度。

从Scrum面板上整理好所有已经完成的用户故事，逐个解释每个故事的功能范围以及该故事为整个项目的业务所提供的价值。故事所属特性实现之后的表现就是应用行为上有了变化。接下来就需要在实际系统上部署产品来为客户展示这些变化了的行为。要欢迎与会者的提问，但是要防止出现太多偏离主题或无关的讨论。要保持整个讨论始终聚焦在正在展示的用户故事上，并在演示结束后再与提出问题的人作进一步深入讨论。会上收集到的所有改进的建议和意见都要加入到产品积压工作中，这样才可以作进一步排序和计划。有时也会发现演示会议上收集到的某些改进建议的优先级会非常高，但这种情况在实际项目中难得一见。

不要害怕演示，演示肯定会激励后续的进度。没有人会在什么都没做的情况下就去中止演示，但是切忌绕开完成的定义去做演示。坦诚公开进度，就不必隐藏任何问题。演示出现问题时，根据图表和度量标准来解释可能的原因即可。

在演示之前的某个时间点冻结代码也是个好主意，这样可以防止开发人员最后时间为了多争取一些故事点数而仓促修改。会议之前要预留合理的时间来搭建演示环境和进行预演，不要为了眼前的一些改进而稀里糊涂地给代码引入技术负债。

> 自律是一种始终能够拒绝眼前诱惑而选择长期利益的能力。
>
> ——Mike Alexander，健身专家

1.4.5　冲刺回顾会议

冲刺演示会议后，需要对整个迭代过程做复盘来评估整个冲刺的成功程度。有些团队成员可能在这个冲刺中取得了很好的战果，而有些人却可能经历了一个糟糕的冲刺。冲刺回顾会议有助于团队从冲刺过程中整理出那些需要保持的好行为以及那些应该避免的不好行为。冲刺回顾会议的输出文档不应该在会后就被束之高阁。应该在下一个冲刺结束时用它作对比，看看哪些需要的改进完成了，哪些错误没有再反复。

会议期间，团队应该回答以下问题。

- 做得好的标准是什么？
- 做得差的标准是什么？
- 需要开始做什么？
- 需要停止做什么？
- 需要继续做什么？
- 是否遇到任何意料之外的事情？

首先从积极的方面开始，让每个团队成员都说说冲刺中哪些地方做得比较好。大家也许对冲刺进度很满意，或者对工作质量很自豪。

接下来，每个团队成员需要讲讲冲刺中哪些地方做得比较差。也许是有些任务延迟交付了，因为任务实际比评估时看起来要困难得多。无论是什么问题，肯定都可以找到解决方案。只要对事不对人，所有有关问题的畅所欲言都没错。团队成员之间不能出现指责，要以恰当的方式建设性地指出问题。讨论问题的目标只是为了改善流程和产品。

团队很可能有一些事情没做但应该在后面加入到流程中。也许是代码还没有正式的单元测试，团队成员都认为应该在现阶段引入单元测试。Scrum主管应该在回顾会议上及时记录所有的建议以便在后期采取行动。

同样地，团队也很可能会发现有些事情不能再继续。很典型的例子就是，在会议上讨论不要跑题，不要在开发泳道已经饱满的情况下还加入新的故事。后面这个问题很常见，可以通过给特定的泳道增加容量限定来解决。通过强制团队最多同时开发三个故事，可以鼓励团队尽快完成已经开始的工作，而不是再开始一些新的工作项。

有些重复性的工作做好了也会有收益。如果因为准备充分取得了不错的冲刺演示效果，那就应该记下来并在以后继续这个好习惯。让人非常意外的是，通常人们会很快忘掉好习惯，而坏习惯却很容易如影随行。

最后，团队成员要一起回忆冲刺期间是否遇到了一些意料之外的事情，不论好或坏。对于不好的意外事件，要讨论出一个行动计划以避免将来出现同样的问题，而好的意外事件能让团队知道有些行为是好的习惯并应该继续坚持。

最后团队成员还需要一起讨论并为下一个冲刺的候选工作项排列优先级。冲刺回顾会议输出的文档不应该在会后就抛之脑后，而是应该根据文档记录内容尽快采取行动。

故事点数三角测量法

冲刺期间，团队成员可能发现有些用户故事的实际工作量与冲刺计划会议上的估算有些或多或少的偏差。那么在冲刺回顾会议结束前，花上5到10分钟，根据故事实际工作量，应用故事点数三角测量法来重新评估用户故事的估算是有好处的。

若干个冲刺后，团队会得到一组统计数据，可以对比每个故事实际工作量和估算点数。举个例子，也许团队会得到如表1-1所示的数据。

表1-1　一个假定项目的实际工作量统计数据和用户故事估算数据的对比表格

故事点数	平均实际工作量（小时）	最小实际工作量（小时）	最大实际工作量（小时）
1	5.5	1	19
2	9.5	2	23
3	17	7.5	40
5	36	20	76
8	56	40	86
13	88	68	154

根据上面表格的统计数据，如果一个用户故事估算为一个点数，而实际花费了六十个小时，那么这个故事很可能应该是一个八个点数的故事。如果开发情况并没有好转（比如缺席的开发人员还没有回来），那么就完全可以认为现有的一个点数的估算是不正确的。相反，如果估算这个故事为八个点数，那么团队的工作速度不会受到影响，团队能承诺的故事总数也不会减少。[①]

此外，只需要关注那些明显偏离的故事估算。因为如果一个点数的故事实际工作量在两个或三个点数的实际工作量范围内时，它的实际工作量也不太可能再发生更大的偏差。

1.4.6　Scrum 日历

为了清楚起见，表1-2中的日历展示了一个冲刺期间所有典型的Scrum会议。

---

① 假设原有一个点数的故事是三个开发人员一起做，结果其中两个人因事缺席后，剩下的一个人最终没有在一个冲刺中完成该故事，那么这个冲刺团队承诺的故事数就会少一个。结合前几个冲刺完成的故事数目求取平均值得到的团队速度也会下降。作者在这里想描述这种因主观或客观原因导致估算不正确的情况，上面这种实际和估算数据对比的统计表格可以给团队后续的估算一些指导和参考，以减少这种偏差对项目的影响。——译者注

表1-2 一个假定项目的Scrum日历，用来组织一个冲刺中的所有会议

日期（2013年4月）	时 间	会议类型	与 会 者
4月2日，周二	13:00～15:30	冲刺计划会议	开发团队 产品负责人
4月3日，周三	09:30～09:45	每日站立会议	开发团队
4月4日，周四	09:30～09:45	每日站立会议	开发团队
4月5日，周五	09:30～09:45	每日站立会议	开发团队
4月8日，周一	09:30～09:45	每日站立会议	开发团队
4月9日，周二	09:30～09:45	每日站立会议	开发团队
4月10日，周三	09:30～09:45	每日站立会议	开发团队
4月11日，周四	09:30～09:45	每日站立会议	开发团队
4月12日，周五	09:30～09:45	每日站立会议	开发团队
4月15日，周一	09:30～09:45	每日站立会议	开发团队
4月16日，周二	10:00～11:20	冲刺演示会议	任何人
	11:30～12:00	冲刺回顾会议	开发团队
	13:00～15:30	冲刺计划会议	开发团队 产品负责人

通过上面的Scrum日历可以看出，一个冲刺的演示和回顾会议以及下一个冲刺的计划会议几乎需要一整天来完成。这一天也被称为冲刺交接日，用来维护冲刺的关注点，有时候这种交接也会分布在前后两天：一个下午和第二天的上午。上面表格中，如果冲刺演示和回顾会议被安排到周二的下午，新冲刺的计划会议就要移到周三的上午。

日历中另外值得注意的一点是，每日站立会议的时间问题。如果安排的时间太早，与会者可能会因堵车或者其他事由迟到。同样，如果安排的时间太晚，要把与会者从紧张的工作中拉出来也会不容易。

可以把这些会议都加入到Microsoft Outlook或其他日历应用中，并将所有相关人指定为与会者，这样就可以保证所有人都能看到相同的会议日程以及提醒。

1.5 Scrum 和敏捷的问题

敏捷流程并不是保证能够挽救所有失败项目的仙丹良药。任何软件开发流程的目标都是每次都能保证成功交付软件，但是软件依然需要编写代码才能实现。任何文档都无法掩盖软件产品是基于可工作代码的事实。

本书的目的是为开发人员讲解如何创建自适应的软件方案。这意味着这个方案要能很好地适应所有软件都遇到的各种变更。寄希望于方案的第一次尝试就能满足客户的所有需求是不现实的，因此变更是无法避免的。敏捷流程（也包括Scrum）的目标都是拥抱变化，并寻求各种方式来确保客户能够对已经实现的软件行为提出变更。没有这些流程，用户可能只好不得不接受一个无法达成目标的不合格方案。

僵硬的代码

不能适应变更的代码就是僵硬的代码，反过来讲，僵硬的代码也很难响应变更。分配给团队的各种任务的估算会与实际情况有明显的偏差，因为编写代码花费了太多的时间，而这是不应该的。后续对僵硬代码的改动还可能引入很多缺陷，修复它们需要更多时间、精力和资源。

1. 死板

死板的代码会有一些症状表现，必须完全解决它们，否则代码会越来越难改变，能交付的特性数目也会越来越少。

- 缺少抽象

抽象能用更简洁的表达来隐藏细节信息。我们周围的抽象示例比比皆是。比如汽车的方向盘可以轻松地引导汽车转向，它抽象了复杂的机械实现。实际上，有两种常见的转向方式：齿轮齿条转向和循环球转向。无论是哪种方式的实现，最终的结果都是轮子按照各自的方式随着方向盘的转动而转向。左右轮子的转向量也不一样，内部轮子的转向半径要比外部轮子小，因此内部轮子的转向角度一定要比外部轮子大一些。

当然，开车并不需要知道这些细节。虽然知道这些有助于诊断问题或者解释工作原理，但是对于普通的驾驶员来讲，这些无关驾驶的信息都不重要。抽象对内隐藏了尽可能多的细节，对外只展示恰好够用的信息。

抽象在软件中同样非常关键。比如，用户界面无需知道存放用户输入的存储介质类型。实际上，如果界面知道细节，这就说明软件的实现缺乏抽象，这个用户界面会因为带有自己不需要且不应该关心的细节而变得很难使用。

具有足够抽象的代码会更容易组织、理解、与其他代码通信、维护，而且错误也更少。

- 职责不清

通常，代码规模都是有组织地从小变大，或从细小琐碎变得庞大重要。随着更多变更的加入，代码也在一层一层地增加，一直到“压死骆驼的最后一根稻草”出现，某一次的改动最终导致了一连串相关且不可预计的问题。

这些代码实现的方法、类型定义甚至整个模块都没有明确的目的。相反，这些代码实现了多个不同的职责却又无法轻易分开。面对这种代码，本来只需要几个小时就可以完成的变更却需要耗费一天甚至更多的时间，因为这种代码中的一个改动可能会引起一系列表面上看起来无关的副作用。

为了避免这种混乱的现象，必须要确保每个层次（方法、类型定义以及程序集）的代码实现都要基于单个明确的职责定义上。

2. 不易测试

单元测试是从很多年前就开始出现的编码实践。对于很多开发人员来讲，单元测试很自然地是一个确保代码正确性的可靠方法。不管怎样，开发人员都需要付出持续的努力才可以长期确保代码是可测试的。

如果代码无法测试，那么它一定就是没有经过测试的，如果代码没经过测试，那么它就一定

有缺陷。这里不需要什么数据来证明，你必须假设未经测试的代码是有缺陷的。这是面对未经测试代码时应该持有的怀疑态度。

下面会简要介绍几个概念以及易测性，后面第4章会作进一步的详细讨论。

- 天钩与塔吊

下面这段文字摘抄自Daniel C. Dennett于1995年撰写的*Darwin's Dangerous Idea: Evolution and the Meanings of Life*一书（我将其中的核心概念设置为粗体）。

> **天钩**（skyhook）……是生命进化基本原则的一个例外，它所产生的设计全都是无任何明确动机的突变产物，与此相反，**塔吊**（crane）则是该生命特征设计流程的一个特别的特征或是其一个子流程，它本身能够加快基本且缓慢的物竞天择的过程，同时也是基本进化流程的一个可以预见的（或者说，事后可以清楚解释的）产物。

简单地说，天钩会直接绕过现有的所有生命特征，它可以用来解释那些跟祖先毫无关系的突变。相反，塔吊则与祖先有着明确的遗传关系，也许直到某个天钩的出现才会变得不明显。

这个生命进化理论上的类比在编码中也同样有用。天钩的出现表明有了深层次的问题发生。应该把所有天钩替换为合适的塔吊。

天钩在代码中很难用模拟实现替换，因此也降低了代码的易测性。天钩的示例包括以下这些。

- 静态方法
- 静态类型（包括单例）
- 使用new的对象创建
- 扩展方法

这些例子都会妨碍给代码注入模拟的替代实现，从而让测试变得更困难[①]。出现天钩的地方都会显得没有任何根据。

幸运的是，上面的天钩都有对应的塔吊，而塔吊代码都可以从外部注入，与天钩不一样，出现塔吊的地方都是有所依据的。塔吊的示例包括以下这些。

- 接口
- 依赖注入
- 控制反转
- 工厂

本书后几章会逐个详细介绍这几个编码“塔吊”。

3. 度量标准

多年以来，已经有了很多不同的源代码度量标准，它们都尝试量化并降低代码的复杂性，从而评测代码或整个项目的健康程度。

这些度量标准似乎在实际项目中用得不是很多，但是自从中层管理者喜欢上源代码行数

① 虽然困难但也有可能做到。诸如TypeMock（http://www.typemock.com）等模拟注入框架能够做到使用模拟实现替代代码中的天钩实现。然而，只有在使用第三方的不可改变的代码库时，才应该考虑使用这种注入框架替换其中的天钩实现。

（source lines of code， SLOC）后，源代码度量标准也多少取得了一些发展。尽管SLOC能很好表现所需的工作量的大小（代码量大一些，耗时自然会多一些），但是它并不能代表系统功能的强弱。同样，用源代码行数评测开发人员的工作编码效率也不可靠。

后发者因之而变（cum hoc ergo propter hoc）

这句拉丁名言的意思是："发生时有这个，因此是因为这个。"它本身是逻辑悖论（推理中出现自相矛盾的错误）的一个例子。因为根据统计事件A发生时总是与事件B有关联，所以事件A的发生都是由事件B引起的，这个推理实际上是一种误解。正确的看法有时候会被引述为"有关联并不代表有因果关系"。

请记住，所有这些代码度量标准希望创造的价值与泛泛的"好代码"的目标之间仅仅只是关联关系，并不是说应用了这些度量标准，就一定能得到真正的"好代码"。

- 单元测试覆盖率

单元测试覆盖率（unit test coverage）是用来度量代码有单元测试覆盖的百分比。百分之零表示所有代码都没有任何测试，百分之百表示每一行代码都至少被一个单元测试覆盖到了。通常认为单元测试覆盖率至少要达到百分之八十。

除了单元测试本身外，还应该将单元测试覆盖率检查工具加入到持续集成流程中。持续集成系统会在每次看到新代码提交时做一次全新的编译，从而快速为开发人员反馈单元测试覆盖率的检查结果。

因为测试覆盖率是对单元测试的定量度量，而非定性的，因此它在实际应用中多少会产生一些误解。比如通过增加任何测试就能轻易地提高代码覆盖率，但是这样并不能保证增加的测试都是正确的。

如果测试覆盖率低于百分之八十（或者是团队设定的其他标准），随着更多新代码的加入，整体覆盖率目标也会受到影响。因此，每次增加新代码时，如果持续集成构建过程发现覆盖率低于设定的标准，就应该给出构建失败的警示。这意味着新加入的产品代码必须要有相应的单元测试，否则这些代码根本无法通过持续构建流程的检查，因为它们会导致整体测试覆盖率低于所设定的标准。

- 圈复杂度

圈复杂度（cyclomatic complexity）是用来度量代码中路径分支的数目。代码中每增加一个分支（`if`语句、循环或`switch`语句等），圈复杂度也会相应增加。图1-12展示了一个简单的`if`语句及其内嵌循环的有向图，其中的代码路径数目就等于代码的圈复杂度。

箭头连线1表示`if`语句进入`false`分支，此时`if`条件为`true`的代码块没有被执行。箭头连线2表示`if`语句进入`true`分支，但未执行内嵌循环。箭头连线3表示`if`语句进入`true`分支并执行了内嵌循环。

随着复杂度的增加，需要为新的代码分支编写更多单元测试来保持测试代码覆盖率。因此，尽量不要增加分支（也就是复杂度）以避免额外的测试代码工作。

此外也有统计数据表明了高复杂度和缺陷数目的关系：分支越多的代码缺陷也越多。

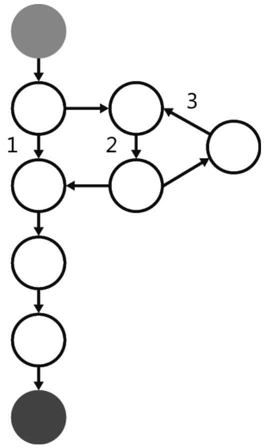


图1-12　代码中每条路径都增加了复杂度

1.6　总结

本章对Scrum流程进行了介绍。如果你以前没有实践过Scrum，我希望你已经跃跃欲试地准备实践Scrum。如果你已经在参与某个Scrum项目，或许会想在手头的项目中实践一下本章中提到的一些新的想法。

本书剩余章节将更多地从开发人员角度讲解敏捷项目的一些最佳实践，而本章肯定有很多有关Scrum的方面没有涉及，不过不要紧，现在有很多公开的学习Scrum的资源，大家可以从中为自己的公司和项目甄选合适的实践。

与任何软件项目一样，Scrum项目也容易受到失败的影响。定位问题很重要，但是理清问题发生的根源更困难。代码的内部策划的设计如果很死板，此时无论使用什么流程来管理变更都一样难以响应变更。本书剩余章节会讲解到一些建议和指导原则，应用它们可以确保代码是自下而上自适应的，让变更响应更容易，并能让开发工作完全聚焦在为每个冲刺增加业务价值的行动上。

第2章 依赖和分层

完成本章学习之后，你将学到以下技能。

- 管理从方法层到程序集层的复杂依赖关系。
- 识别最复杂的依赖关系并使用工具降低复杂度。
- 将代码拆分为更小的、更有适应能力的、更易重用的功能代码块。
- 在最恰当的地方应用分层模式。
- 清楚如何解决依赖以及调试依赖问题。
- 使用简洁的接口隐藏实现。

所有软件都有依赖。这些依赖要么是对同一个解决方案内的第一方程序集的依赖，要么是对第三方外部程序集的依赖，或者是处处可见的对Microsoft .NET Framework的依赖。几乎所有稍微有些规模的项目都会存在这三种依赖关系。

依赖项抽象了你编写的客户端代码要使用的功能。你不需要知道依赖在做什么，更不需要知道依赖内部是怎么做的，但是应该确保正确管理所有的依赖。如果不能很好地管理依赖链，开发过程中就会很容易引入根本就不需要的依赖，结果就是代码与很多不必要的程序集紧紧地纠缠在一起。你也许听过这么一句老话："没编写的代码就是最正确的代码。"同样，不存在的依赖就是管理得最好的依赖。

为了让代码能够自适应变更，你需要高效管理所有的依赖关系。这对软件的所有层次，从架构层子系统间的依赖到实现层每个方法之间的依赖，都是必要的。糟糕的架构会拖延可工作软件的交付，甚至导致项目夭折。

我认为无论怎样强调高效管理依赖的重要性都不为过。如果在重要的问题上采取了折中的临时方案，也许短时间内可以提高开发效率，但是长期的副作用很可能会对项目造成致命的伤害。下面是一个我们再熟悉不过的场景：一开始，随着代码和模块数量的增加，短期的开发效率非常高。慢慢地，代码变得死板和混乱，由此进度也变得缓慢甚至停滞。用Scrum的术语描述就是，在发现根本问题前，缺陷数目在增加且无法获得故事点数，也就无法完成任何故事和特性，因此冲刺燃尽图和特性燃耗图也会没有一丝变化。当依赖结构混乱且无法理解时，一个模块的变更很可能会给另外一个看起来无关的模块带来副作用。

要想轻松管理依赖，需要有清醒的认识并按照指导原则行动。应用一些现有的模式可以帮助代码适应后期的变更。分层就是一种最常见的架构模式，本章会详细讲解几种不同的分层方法，

此外也会介绍其他一些依赖管理方法。

2.1　依赖的定义

什么是依赖？通常来讲，依赖（dependency）是指两个不同实体间的一种联系，如果没有其中一个，另一个就会缺少某些功能甚至会不存在。一个比较好的类比是，某个人在财务上对另外一个人有依赖。通常的法律文件都需要你声明是否有家属，也就是说，是否有人需要你来承担他们的生活和其他必需品的费用，家属通常是指你的配偶和孩子。举个我的例子，当我在英国的百慕大工作时，我的工作许可证上写着："一旦我的工作许可证失效，我的妻子和女儿就需要和我一起离开工作地。"这种情况下，她们作为我的家属需要依靠我在当地生活。

在代码上下文中看依赖的定义，实体通常是指程序集。程序集A使用程序集B，就可以说A依赖B。这种关系的一种常见说法是：A是B的客户（client），B是A的服务（service）。没有B的话，A无法起作用。然而，B并不依赖A这一点也很重要，接下来你会学到，B不可以也不能依赖A。图2-1展示了这种客户/服务关系。



图2-1　在依赖关系中，依赖者称为客户，被依赖者称为服务

本书全篇都是以客户端和服务端的视角来讨论代码的。有些服务是在远程主机上，比如使用Windows Communication Foundation（WCF）创建的服务。无论是否为远程代码，都可以称为服务。代码是服务端代码还是客户端代码取决于你看待代码角色的角度。任何类、方法或程序集都可以调用其他方法、类和程序集，因而代码是客户端代码。相同的类、方法或程序集也可以由其他方法、类和程序集调用，因而代码也是服务端代码。

2.1.1　一个简单的例子

我们来看看具体程序中的依赖行为。这个例子就是著名的"Hello world"范例，这个简单的控制台程序只负责打印一句消息。我选择这么简洁的例子是为了能更清楚地展示程序中存在的依赖问题。

你可以按照下面步骤创建示例，也可以从GitHub上直接下载源代码。有关如何使用Git的基本介绍请参见附录。

(1) 打开Microsoft Visual Studio，创建一个控制台程序。我把它命名为SimpleDependency，这里名称不重要，可以随意更改。如图2-2所示。

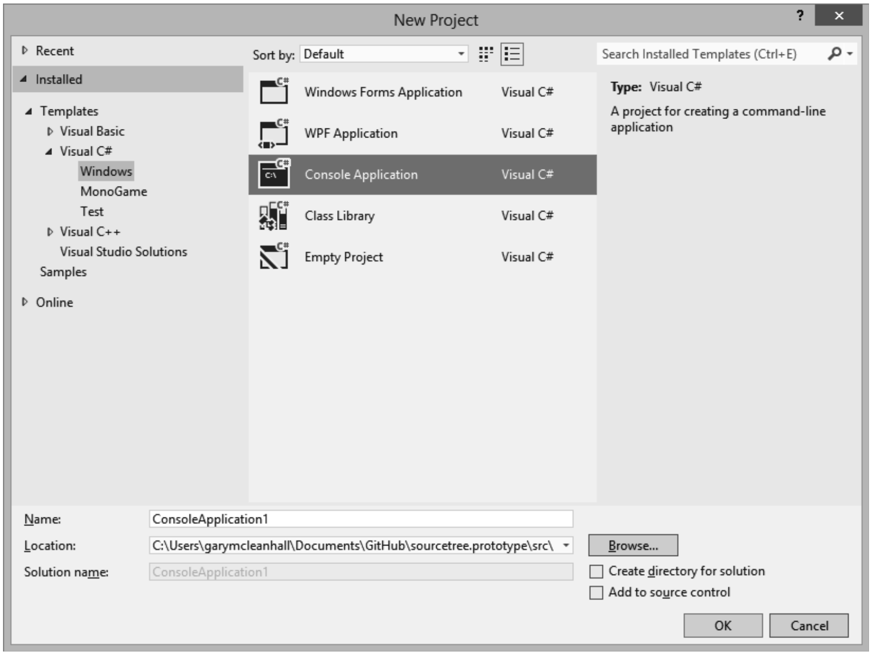


图2-2　Visual Studio的新建项目对话框中有很多不同的项目模板可选

(2) 给解决方案再添加一个新的类库项目。我的命名是MessagePrinter。

(3) 右键单击控制台应用的References（引用）节点并选择Add Reference（添加引用）。

(4) 在弹出的Add Reference对话框中，导航至Projects（项目），然后选择类库项目 。

如图2-3所示，两个程序集之间现在有了依赖关系。控制台程序依赖类库，但是类库并不依赖控制台程序。控制台程序是客户，类库是服务。这个应用还没有什么功能，先选择构建解决方案，然后打开SimpleDependency项目下存放可执行文件的bin目录。

bin目录不仅包含了SimpleDependency.exe文件，还包含MessagePrinter.dll文件。构建解决方案的过程中，Visual Studio会自动将MessagePrinter.dll文件复制到SimpleDependency项目的bin目录下，因为它发现项目MessagePrinter被项目SimpleDependency引用为一个依赖项。我想用这个示例给大家展示一个实验过程，不过要先对控制台程序做些小小的修改。因为这个控制台程序什么都没做，它运行起来后会马上直接退出。现在打开控制台程序的Program.cs文件。

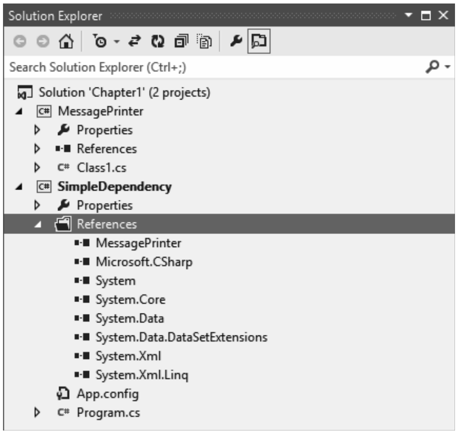


图2-3　任何项目的References节点下都会列出该项目的引用程序集

代码清单2-1展示了在Main方法内增加的代码（加粗）。Main方法是控制台应用的入口，修改前它没有任何动作并且会直接退出。通过增加调用Console.ReadKey()，可以让应用一直保持运行状态直到有任何键盘输入。

代码清单2-1　通过调用Readkey来阻止控制台程序立即退出

```
namespace SimpleDependency
{
    class Program
    {
        static void Main()
        {
            Console.ReadKey();
        }
    }
}
```

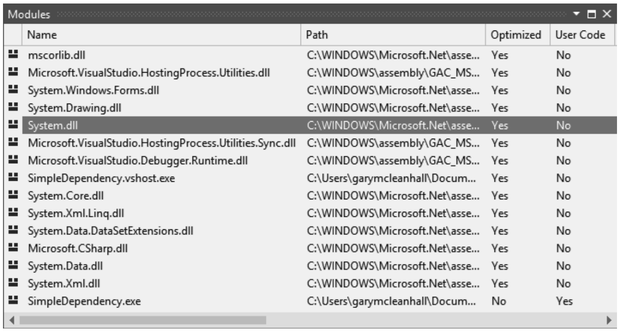
修改代码后重新构建解决方案，然后运行应用程序。与期望的一样，应用程序会显示控制台窗口，并一直等待键盘输入直至退出。使用Visual Studio在代码行Console.ReadKey()前设置断点，然后选择调试运行。

当程序运行到达断点时，你能看到为该应用程序加载到内存的程序集列表。Visual Studio有两种方式可以查看：使用菜单栏依次选择Debug（调试）> Windows > Modules（模块），或者直接使用快捷组合键Ctrl+D+M。图2-4展示了为该应用加载到内存的模块列表。

在图2-4中，你有没有看到一些奇怪的地方？列表中并没有包含你刚刚创建的类库。在这个例子里，不是应该将MessagePrinter.dll文件加载到内存中吗？实际上，不应该加载，而且这也的确是期望的正确行为。原因是：应用程序并没有使用MessagePrinter程序集内的任何功能，所以.NET运行时并不会加载它。

为了进一步证明项目中引用的MessagePrinter程序集并非真的所需，可以直接到应用程序的

bin目录下删除MessagePrinter.dll文件。然后再次运行程序，结果一切正常，并不会看到任何异常发生。



Name	Path	Optimized	User Code
mscorlib.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
Microsoft.VisualStudio.HostingProcess.Utilities.dll	C:\WINDOWS\assembly\GAC_MS...	Yes	No
System.Windows.Forms.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.Drawing.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
Microsoft.VisualStudio.HostingProcess.Utilities.Sync.dll	C:\WINDOWS\assembly\GAC_MS...	Yes	No
Microsoft.VisualStudio.Debugger.Runtime.dll	C:\WINDOWS\assembly\GAC_MS...	Yes	No
SimpleDependency.vshost.exe	C:\Users\garymcleanhall\Docum...	Yes	No
System.Core.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.Xml.Linq.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.Data.DataSetExtensions.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
Microsoft.CSharp.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.Data.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
System.Xml.dll	C:\WINDOWS\Microsoft.Net\asse...	Yes	No
SimpleDependency.exe	C:\Users\garymcleanhall\Docum...	No	Yes

图2-4　调试时，Modules窗口会显示所有当前已经加载的程序集

我们再重复几次这个实验，看看到底会发生什么。首先，在Program.cs文件顶部加入`MessagePrinter`命名空间的`using`指令。你认为这样公共语言运行时（Common Language Runtime，CLR）就会加载这个模块吗？答案还是否定的。公共语言运行时会再次忽略这个依赖项，也不会加载这个程序集。这是因为用于导入命名空间的`using`语句只是一个语法糖，它的设计目的只是为了减少你编写代码的工作量。当你需要使用命名空间中的任意类型时，只需要导入命名空间然后直接引用这些类型定义，而不需要在类型名前带上完整的命名空间。因此，编译`using`语句并不会生成公共语言运行时要执行的指令。

在上面实验的基础上，保留Program.cs文件顶部的`using`语句，然后在`Console.ReadLine()`调用前再加入一个对`MessagePrintingService`构造函数的调用。如代码清单2-2所示。

代码清单2-2　通过调用一个实例方法来引入依赖关系

```
using System;
using MessagePrinter;

namespace SimpleDependency
{
    class Program
    {
        static void Main()
        {
            var service = new MessagePrintingService();
            service.PrintMessage();
            Console.ReadKey();
        }
    }
}
```

这一次调试时的Modules窗口显示MessagePrinter.dll程序集已经被加载了，因为如果不把该程序集内容加载到内存，就无法创建`MessagePrintingService`类的实例。

如果想作进一步确认，可以尝试删除bin目录下的MessagePrinter.dll文件并再次运行应用程序。这次你会看到应用程序引发了一个如下的异常。

```
Unhandled Exception: System.IO.FileNotFoundException: Could not load file or assembly
'MessagePrinter, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null' or one of its
dependencies. The system cannot find the file specified.
```

1. 对.NET Framework的依赖

上一部分展示的依赖被称为第一方（first-party）依赖。控制台应用程序和它所依赖的类库都在同一个Visual Studio解决方案下。这就意味着第一方依赖总是可用的，因为在需要的时候，被依赖的项目总是可以通过源代码重新构建。也代表你能够直接修改第一方依赖的源代码。

这个例子中的两个项目都依赖其他一些.NET Framework程序集。它们并不是项目本身的一部分，但是依然需要它们是可用的。每个.NET Framework程序集都带有自己的目标框架版本：1、1.1、2、3.5、4、4.5等。有些程序集只属于某个新版本的.NET Framework，无法被使用早期框架版本的项目引用。剩下的程序集虽然在每个版本的.NET Framework都有，但是不同版本之间是有功能差别的，所以在使用时需要指定具体的版本号。

如前面的图2-3所示，SimpleDependency项目对.NET Framework有多个引用。这些依赖中很多都是Visual Studio默认加入到所有控制台应用程序项目中的。这个示例控制台应用程序并没有使用它们的任何功能，因此可以放心地移除对它们的引用。实际上对于这个例子中的两个项目，除了System和System.Core之外，其他的.NET Framework程序集都是多余的，可以把它们从引用列表中删除。删除后，程序仍然可以正常运行。

通过移除不必要的对.NET Framework的依赖，你会更容易看清每个项目必需的依赖清单。

框架程序集总是会加载

不像其他依赖，对.NET Framework程序集的引用总是会导致加载这些程序集。即使你并没有真正使用某个.NET Framework程序集，它依然会在应用程序启动的时候被加载到内存中。幸运的是，如果同一个解决方案下的多个项目都在引用同样的.NET Framework程序集，在运行时，所有依赖它们的项目会共享同一份加载到内存中的.NET Framework程序集实例。

- 默认的引用列表

Microsoft Visual Studio中，不同的项目类型有不同的默认引用清单。每个项目类型都有一个项目模板，其中包括了需要的引用清单。Windows Form应用程序的模板中默认指定的引用包括了System.Windows.Forms程序集，而Windows Presentation Foundation（WPF）应用程序的引用则包括了WindowsBase、PresentationCore和PresentationFramework这几个程序集。

代码清单2-3展示了控制台应用程序的默认引用清单。Visual Studio把所有项目类型的模板文件都存放在安装目录下的子目录/Common7/IDE/ProjectTemplates/下，其中每个语言都有对应版本的模板文件。

代码清单2-3 Visual Studio项目模板的一个片段，可以根据条件引用不同的程序集

```
<ItemGroup>
    <Reference Include="System"/>
    $if$ ($targetframeworkversion$ >= 3.5)
    <Reference Include="System.Core"/>
    <Reference Include="System.Xml.Linq"/>
    <Reference Include="System.Data.DataSetExtensions"/>
    $endif$
    $if$ ($targetframeworkversion$ >= 4.0)
    <Reference Include="Microsoft.CSharp"/>
    $endif$
    <Reference Include="System.Data"/>
    <Reference Include="System.Xml"/>
</ItemGroup>
```

类似于上面的代码清单2-3，在所有项目模板中都会根据不同情况创建实际的项目实例。特别是，使用不同版本的.NET Framework的项目会引用不同的程序集。上面的例子中，我们可以看到，只有当项目使用.NET Framework 4、4.5或4.5.1时，项目实例才会引用Microsoft.CSharp程序集。这样做的意义是，只有使用从.NET Framework 4引入的`dynamic`关键字时，你的项目才需要引用这个程序集。

2. 第三方依赖

最后一种要依赖的是由第三方开发的程序集。通常，第三方程序集不是由.NET Framework提供的，你也可以选择方案并直接实现为第一方程序集。有时候，如果方案的规模比较大，自己编码实现的工作量也会比较大，此时，可以选择现有可用的方案实现。举个例子，你很可能不会想去自己重新实现诸如对象/关系映射器（Object/Relational Mapper，ORM）之类的大型解决方案，因为首先你可能需要好几个月来编码，然后可能再耗费好几年时间来完成全部测试。这种情况下，你应该首先看看.NET Framework提供的Entity Framework是否够用，如果它不能满足你的需求，你还可以使用NHibernate，它是一个经过了广泛测试的、成熟的对象/关系映射器库。

只需集成现有可用的、满足需求的特性或基础构件的实现，而无需完全重新实现它们，这是使用第三方依赖的主要原因。当然，集成工作的难度也可能会比较大，这个取决于你的第一方代码以及第三方代码接口的结构。当你需要使用迭代方法以增量方式（就像Scrum流程）发布有业务价值的特性时，使用第三方程序集能够帮助你聚焦在业务相关的工作重点上。

- 组织第三方依赖

最简单的组织方式就是直接在项目解决方案下创建一个名为Dependencies的解决方案文件夹，然后把所有的第三方.dll文件存放在这个文件夹下。当项目需要引用这些程序集时，你只需要使用引用管理器浏览这个文件夹（如图2-5所示）。

这种方式的优点是所有外部依赖都存放到了源代码控制平台上。其他的开发人员从中心代码库中获取最新源代码时也能获得最新的依赖程序集，这样就不需要所有开发人员单独下载和安装这些要依赖的程序集。

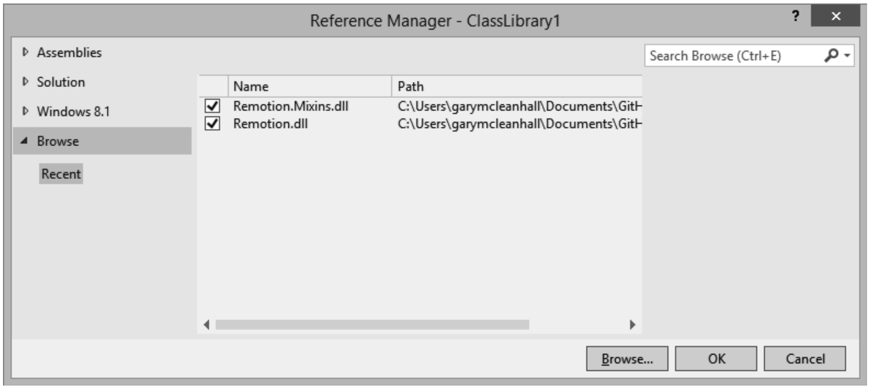


图2-5　可以将第三方引用存储在Visual Studio解决方案下的Dependencies文件夹中

本章稍后的2.2.7节会讲解一个更好的组织第三方依赖的方式。简单来讲，NuGet依赖管理工具能够为开发人员自动管理项目的第三方依赖，它可以下载依赖安装包（包括相关文档），引用程序集，以及将依赖库升级到最新版本。

2.1.2　使用有向图对依赖建模

图（graph）是一种数学建构，它包括两种元素：节点和边线。边线只能用于连接两个节点，代表了两个节点之间的某种联系。图中的任一节点可以与其他节点通过边线连接起来。不同属性的图所属的种类不同。比如，图2-6中展示的图是无向图（undirected graph）。这个图中，节点之间的边线是没有方向的：节点A和C之间的边线可以是从A至C，也可以是从C至A，图中其他边线也一样是无向的。

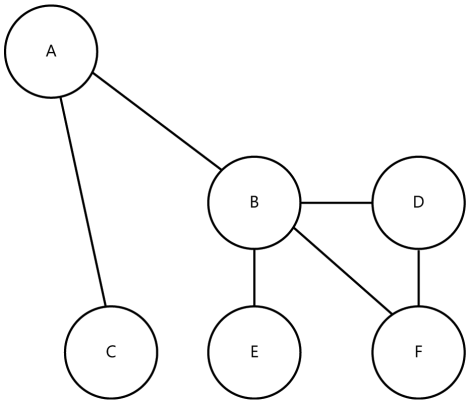


图2-6　图形由用边线连接起来的节点组成

而图2-7展示的图则是有向图（directed graph，即digraph）。其中节点间的边线一端都有代表方向的箭头符号：节点A和C之间边线的方向是从A到C的，而不是从C到A的。

图在软件工程的很多领域都有很好的应用，但是它更适合用来对代码间的依赖关系建模。前面几节已经讲解了，一个依赖关系包含了两个不同的代码实体，它们之间的联系方向是从依赖者到被依赖者。你可以把实体看作节点，并从依赖者到被依赖者的方向绘制有向边线。反复在所有其他实体上应用节点和有向边线的概念进行建模，你就能得到一个完整的依赖有向图（dependency digraph）。

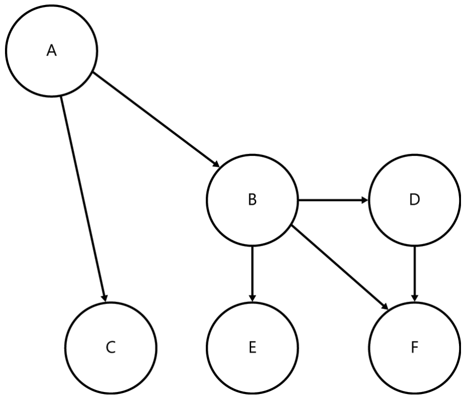


图2-7　这个图中的边线是标明了方向的，所以只有A到B的有向边线，而没有B到A的有向边线

如图2-8所示，可以在项目的不同层次上应用这种依赖关系的建模方式。图中节点可以用来表达项目中的类、程序集或者子系统中的程序集组。无论在哪个层次上，节点间边线的箭头都表达了不同组件之间的依赖关系。箭头从依赖组件指向被依赖组件。

每个大粒度的节点都可以分解成为一组更小粒度的节点。比如：子系统可以分解为一组程序集；程序集可以分解为一组类；类内部仍可以分解为一组方法。图2-8这个示例展示了在整个子系统依赖链中一个单独方法上依赖关系所处的位置。

然而，上面所有的示例只能展示出有依赖关系，但是并没有展示出依赖的分类（比如继承、聚合、复合以及关联）。但是这依然是有用的，因为依赖关系只需要知道两个二进制实体之间的关系：有或没有依赖？

循环依赖

图论中还提到有向图中会形成循环，也就是说从一个节点沿着有向边线遍历后还能够回到这个节点。前面几节中展示的图都没有循环，称为有向无环图（acyclic digraph）。图2-9展示了一个有向有环图（cyclic digraph）的示例。从节点D开始，可以沿着边线经过节点E和B，最后又回到节点D。

如果用节点表示程序集。图中程序集D显式或隐式地依赖一些程序集，因此它也隐式依赖这些程序集所依赖的其他一些程序集。上面图表中，节点D显式依赖节点E，隐式依赖节点B和D，因此节点D也依赖自身。

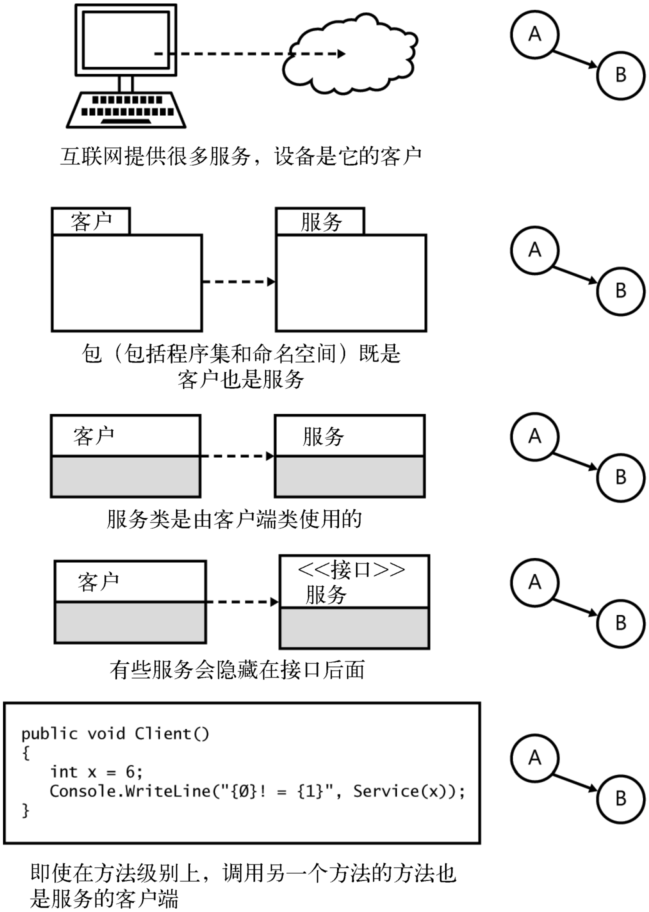


图2-8　各个级别的依赖都可以使用图形来建模

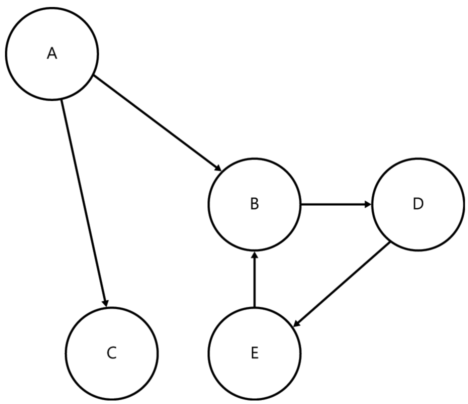


图2-9　这个有向图包含多个循环

如果节点是程序集，这种循环依赖是不可能出现的。不允许在Visual Studio中尝试构建这种循环依赖关系。在项目E中添加对B程序集的引用时，会看到如图2-10所示的警告信息。

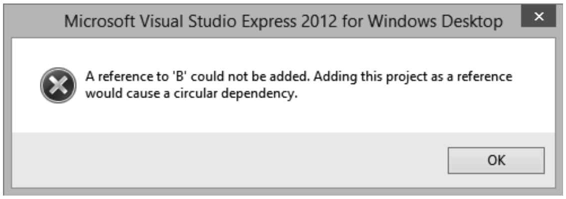


图2-10 不允许在Visual Studio中创建循环依赖关系

尽管使用图对依赖关系进行建模似乎有点太过学术，但是这样组织依赖关系还是很有好处的。理论上，可以存在的循环依赖关系在软件工程的实际应用中完全不允许，而且一定要避免。

有向图中有一种特殊的循环叫作*自循环*（loop）。如果节点通过一条边线直接连接自身，那么这条边线就变成了一个自循环。图2-11展示了带有一个自循环的有向图。

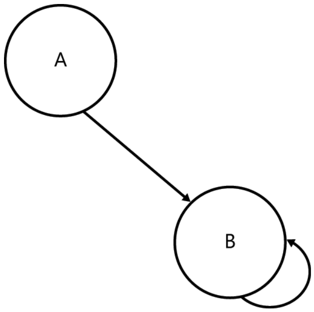


图2-11 这个有向图中，节点B通过一个自循环连接到自身

实际应用中，程序集通常都是显式自依赖的，这一点通常不会被注意到。此外，方法层的*递归*（recursion）就是一个很好的自循环的例子，如代码清单2-4所示。

代码清单2-4 有向图中的自循环可以用来表示递归方法

```
namespace Graphs
{
    public class RecursionLoop
    {
        public void A()
        {
            int x = 6;
            Console.WriteLine("{0}! = {1}", x, B(x));
        }

        public int B(int number)
        {
            if(number == 0)
```

```
                {
                    return 1;
                }
                else
                {
                    return number * B(number - 1);
            }
        }
    }
}
```

代码清单2-4中的类与图2-11中的依赖图表达了相同的功能。方法A调用了方法B，那么方法A依赖方法B。然而，更有趣的是递归方法B，它显式地依赖自身。递归方法就是一个调用自身的方法。

2.2　依赖管理

现在你已经知道，依赖关系是必要的，但是必须小心管理以免产生的问题影响了后期开发。通常，当这些问题暴露出来时，就已经很难解决了。因此，最好从一开始就谨慎正确地管理所有依赖关系，以免不经意间引入问题。糟糕管理的依赖关系引起的微小局部问题会很快升级为项目整体架构上的严重问题。

本章剩余几节会更多地集中讲解如何持续管理依赖（包括避免反模式），更重要的是，要理解为何有些常见的模式是反模式。相反，有些模式是真的有益并值得推广的，它们可以用来替代相应的反模式。

模式和反模式

软件工程历史上，面向对象软件开发算是个相对比较新的尝试。在过去的几十年里，已经有一些类和接口间的协作方法被识别和总结出来，它们是可重用的，并称之为**模式**（pattern）。

软件开发模式有很多种，每种模式都可以在某些特定的问题域重复应用。有些模式协同其他一些模式后还能够为复杂问题提供优雅的解决方案。当然，并不是所有模式总是可应用的，需要花费时间进行实践并积累经验，才能识别出应用这些模式的合适场合。

有些模式并没有多少益处，相反它们实际上还有很多副作用，这类模式被称为**反模式**（anti-pattern）。反模式会破坏代码的自适应能力，应该避免在代码中引入它们。随着很多副作用被发现，一些模式会逐渐被抛弃并归类为反模式。

2.2.1　实现与接口

通常，对于刚刚接触面向接口编程概念的开发人员而言，不让他们去考虑接口后面的实现细

节会很困难。

编译时，接口的所有客户端都不应该知道接口要使用的具体实现。如果知道具体实现，会让人错误地认为客户端代码与接口的这个具体实现是直接关联的。

考虑一个常见的例子：一个类能够向永久存储介质中存放记录数据。为了达到这个目的，正确的做法是定义一个隐藏具体永久存储机制细节的接口。另外切记，不要假设运行时会使用某种具体的实现。比如，不要在代码中将接口转换为任何具体实现。

2.2.2 new 代码味道

接口描述能做什么，接口的实现类则描述如何做。只有类才会涉及实现细节，接口应该对如何实现的细节一无所知。这是因为只有类才有构造函数，构造函数则包含了实现细节。在此基础上会得到一个有意思的结论，那就是，除了一些特殊情况外，凡是出现new关键字的地方都是代码味道（code smell）。

> **代码味道**
>
> 如果某段代码可能存在问题，就可以说有代码味道。这里使用“可能”是因为少量的代码味道并不一定就是问题。反模式总被认为是坏的实践，但代码味道不一样，它们不一定是坏的实践。代码味道可以警告可能有错误发生，需要根据根本原因来判断是否要修正。
>
> 代码味道还可能表明有技术债务存在，而技术债务的修复是有代价的。背负技术债务越久，债务修复就会越难。
>
> 代码味道有很多分类。使用关键字new创建对象实例属于“狎昵关系”。因为构造函数是实现细节，客户端代码调用构造函数会引入意外的（也是不希望的）依赖关系。
>
> 与反模式一样，可以通过重构代码来清除代码味道，重构后的代码有着更好的、更具有适应能力的设计。实现了次优设计的代码可能满足了当前的需求，但是将来还是可能会引起问题。代码重构无疑是一个无法立即产生可见效益的开发任务，因为所有修复问题的重构工作都不会附加有相应的业务价值。然而，与金融债务可能导致支付高额利息类似，技术债务也有可能会逐渐失去控制，进而摧毁良好的依赖管理实践，危及后续的代码设计改进和问题修复。

代码清单2-5展示了使用new关键字的代码味道，两个例子都直接创建了对象实例。

代码清单2-5 一个示例，展示如何通过实例化对象来破坏代码的适应能力

```
public class AccountController
{
    private readonly SecurityService securityService;

    public AccountController()
    {
        this.securityService = new SecurityService();
```

```
        }

        [HttpPost]
        public void ChangePassword(Guid userID, string newPassword)
        {
            var userRepository = new UserRepository();
            var user = userRepository.GetByID(userID);
            this.securityService.ChangeUsersPassword(user, newPassword);
        }
    }
```

AccountController类是从一个假定的ASP.NET MVC应用程序中提取的。我们先抛开该类的实现细节，着重看看这些加粗的不恰当的对象构造语句。这个控制器类的职责是允许用户执行账户查询和其他命令。这个示例中只展示了一个命令：ChangePassword。

代码中有下面一些问题，这些问题是由于两个显式调用new关键字的构造对象实例引起的。

- AccountController类永远依赖SecurityService类以及UserRepository类的具体实现。
- AccountController类隐式依赖SecurityService类和UserRepository类的所有依赖。
- AccountController类很难测试，因为无法用伪实现来模拟和替代SecurityService类和UserRepository类。
- SecurityService类的ChangeUsersPassword方法需要客户端代码先加载好User类的实例对象。

下面会详细剖析这几个问题。

1. 无法增强实现

当你想要改变SecruityService类的实现时，只有两个选择，要么改动AccountController来直接引用新的实现，要么给现有的SecurityService添加新功能。阅读本书之后，你会发现这两个选项都不好。现在，我们先把目标定为AccountController和SecurityService在创建后都不允许再做任何改动。

2. 依赖关系链

SecurityService类也会有自己的依赖关系。代码清单2-5中，加粗的SecurityService类的默认构造函数看起来似乎没有任何依赖。但是，如果SecurityService类的构造函数的实现如下面的代码清单2-6所示呢？

代码清单2-6　SecurityService和AccountController两个类有着同样的问题

```
    public SecurityService()
    {
        this.Session = SessionFactory.GetSession();
    }
```

SecurityService类实际上依赖NHibernate（一种对象/关系映射器）库，它被用来获取一个会话（session）。NHibernate使用会话来表示指向持久关系型存储（比如Micorsoft SQL Server、Oracle或MySQL等）的连接。正如前面所讲的，这意味着AccountController类也隐式依赖

NHibernate库。

再者，如果SecurityService类的构造函数签名也要改变呢？也就是说，如果Security-Service类的构造函数突然需要客户端代码提供数据库连接字符串给会话呢？任何使用SecurityService类的客户，包括AccountController类在内，都必须改动代码来提供连接字符串。再强调一次，一定要避免这种糟糕的变更。

3. 缺乏可测试性

可测试性也非常重要，它需要代码以一定的模式构建。如果不这样做，测试将变得极其困难。不幸的是，在代码清单2-5中，AccountController和SecurityService这两个类都很难测试，因为你无法使用不执行任何动作的模拟实现来替代这两个类的实现。举个例子，在测试SecurityService类时，你并不想建立到数据库的连接。因为连接数据库的过程不仅慢也没有必要，而且还会引入另外一个更高失败率的测试点：连接数据库失败。有几种方法可以在运行时使用模拟实现来替代对这两个类的依赖。诸如Microsoft Moles和Typemock之类的工具可以钩入构造函数中，并确保它们返回的对象是Fakes。但是，这些方法要谨慎使用，因为它们只能治标但不能治本。

4. 更多的狎昵关系

AccountController类的ChangePassword方法会先创建一个UserRepository类的实例，然后通过它获取一个User类的实例。它这样做的唯一原因就是SecurityService类的ChangePassword方法的需要。如果没有传入这个User类的实例，就无法调用该方法，这也表明这个方法的签名设计很糟糕。如果所有客户端代码都需要获取一个User类的实例，这种情况下，SecurityService类就应该在自己内部获取User类的实例。代码清单2-7展示了重构后用于更改用户密码的两个方法。

代码清单2-7 对客户调用SecurityService类代码的一个改进

```
[HttpPost]
public void ChangePassword(Guid userID, string newPassword)
{
    this.securityService.ChangeUsersPassword(userID, newPassword);
}
//...
public void ChangeUsersPassword(Guid userID, string newPassword)
{
    var userRepository = new UserRepository();
    var user = userRepository.GetByID(userID);
    user.ChangePassword(newPassword);
}
```

对于AccountController类而言，这绝对是个改进，但是ChangeUsersPassword方法仍然会直接实例化UserRepository类。

2.2.3　对象构造的替代方法

怎么做才可以同时改进AccountController和SecurityService这两个类，或者其他任何不合适的对象构造调用呢? 如何才能正确设计和实现这两个类以避免出现上节所讲述的任何问题呢? 下面有一些能够互补的方式可供选择。

1. 针对接口编码

你应该做的首要改动是将SecurityService类的实现隐藏在一个接口后。这样AccountController类就会只依赖SecurityService类的接口而不是它的具体实现。第一个代码重构就是为SecurityService类提取一个接口，如代码清单2-8所示。

代码清单2-8　为SecurityService类提取一个接口

```
public interface ISecurityService
{
    void ChangeUsersPassword(Guid userID, string newPassword);
}
//...
public class SecurityService : ISecurityService
{
    public ChangeUsersPassword(Guid userID, string newPassword)
    {
        //...
    }
}
```

下一步就是改动客户端代码来调用ISecurityService接口，而不是SecurityService类。代码清单2-9展示了应用这个重构后的AccountController类的情况。

代码清单2-9　AccountController类现在依赖ISecurityService接口

```
public class AccountController
{
    private readonly ISecurityService securityService;

    public AccountController()
    {
        this.securityService = new SecurityService();
    }

    [HttpPost]
    public void ChangePassword(Guid userID, string newPassword)
    {
        securityService.ChangeUsersPassword(user, newPassword);
    }
}
```

这个示例仍然没有结束，因为依然直接调用了SecurityService类的构造函数，所以重构后的AccountController类依然依赖SecurityService类的具体实现。AccountController类的

构造函数还是会实例化具体的SecurityService类实例。要将这两个具体类完全解耦，你还需要作进一步的重构，即引入依赖注入（Dependency Injection，DI）。

2. 使用依赖注入

这个主题比较大，无法用很短的篇幅讲完。实际上，第9章会专门讲解这个主题，此外还有一些依赖注入的专题书籍。幸运的是，依赖注入并不是很复杂或者困难，所以本节会从使用依赖注入的类的角度来讲解一些基本的要点。代码清单2-10展示了新的对AccountController类的构造函数进行的代码重构。重构后的构造函数代码部分已经加粗显示，重构动作的改动非常小，但是管理依赖的能力却大不相同。AccountController类不再要求构造SecurityService类的实例，而是要求它的客户端代码提供一个ISecurityService接口的实现。不仅如此，构造函数中还加入了前置条件的检查语句，用于防止从securityService参数上传入空值的异常情况。这个检查确保了，在ChangePassword方法中需要使用securityService类的实例时，该实例始终有效且无需处处检查空值。

代码清单2-10　使用依赖注入从AccountController类中移除对SecurityService类的依赖

```
public class AccountController
{
    private readonly ISecurityService securityService;

    public AccountController(ISecurityService securityService)
    {
        if(securityService == null) throw new ArgumentNullException("securityService");

        this.securityService = securityService;
    }

    [HttpPost]
    public void ChangePassword(Guid userID, string newPassword)
    {
       this.securityService.ChangeUsersPassword(user, newPassword);
    }
}
```

SecurityService类也同样需要应用依赖注入。代码清单2-11展示了重构后的类实现。

代码清单2-11　依赖注入是一种很常见的模式，几乎可以不受限制地应用于代码中的任意位置

```
public class SecurityService : ISecurityService
{
    private readonly IUserRepository userRepository;

    public SecurityService(IUserRepository userRepository)
    {
        if(userRepository == null) throw new ArgumentNullException("userRepository");
        this.userRepository = userRepository;
    }
```

```
        public ChangeUsersPassword()
        {
            var user = userRepository.GetByID(userID);
            user.ChangePassword(newPassword);
        }
    }
```

与AccountController类改为依赖一个有效的ISecurityService接口实例一样，SecurityService类也改为依赖一个有效的IUserRepository类的实例（构造函数在检测到传入的是空值时会引发异常）。同样地，通过引入IUserRepository接口，SecurityService类对UserRepository类的依赖也被全部移除了。

2.2.4　随从反模式

随从反模式（entourage anti-pattern）这个名称缘于这样的事实：即使你只想要个简单的东西，也总是会得到包括它在内的很多东西。就像歌星或电影明星一样，他们总是带着他们的随从（跟班或助理之类的人）一同进进出出。这是我为这种模式起的名称，用于更恰当地表达这些并不需要的依赖关系。

随从反模式是开发人员使用针对接口编程时很容易犯的一个错误。无需赘述，这里要直接给出的结论是：接口和接口的依赖项肯定不应该布置在同一个程序集内。

图 2-12 中的 UML 图展示了AccountController示例中包层次上的组织关系。AccountController类依赖ISecurityService接口，后者则由SecurityService类实现。图中的包（.NET程序集或者Visual Studio项目）就是依赖关系中实体的容器。图2-12就是一个随从反模式的例子：接口本身和接口的实现类被布局在同一个程序集内。

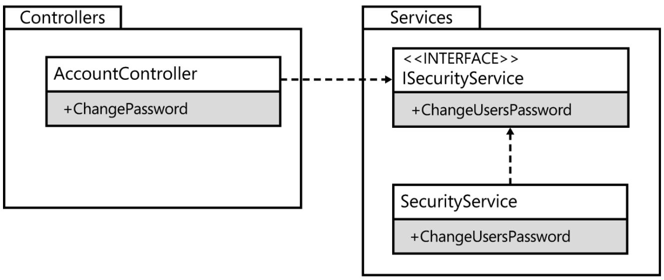


图2-12　AccountController类所在的程序集依赖Services程序集

相信你已经知道，SecurityService类也有自己的一些依赖关系，依赖关系链也会导致客户端之间的隐式依赖。通过扩展包图，图2-13展示了一个完整的随从问题。

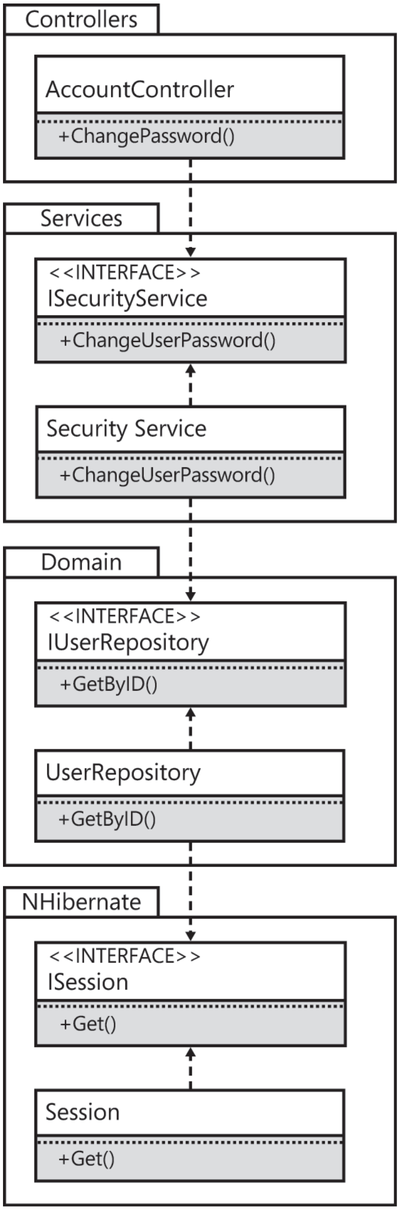


图2-13　`AccountController`类依然隐式依赖了太多的实现

如果你单独构建Controllers项目，会发现bin目录下也会有NHibernate程序集，这表明了Controllers程序集依然隐式依赖了NHibernate程序集。此时，尽管你已经通过多次重构移除了`AccountController`类中那些不必要的依赖关系，但是它与每个要依赖程序集的实现之间依然不是松散耦合的关系。

随从反模式会带来两个问题。第一个问题与开发人员的自律有关。每个包（Services、Domain和NHibernate）都要有设置为`public`的接口定义。然而，你还需要实现具体的类并标记为`public`，因为你总是要在某些地方构造接口实现类的实例（只是不要在客户端类中直接调用）。这就意味着不够自律的开发人员仍然可以直接引用具体实现。很多人都会走“捷径”去直接调用`new`关键字来获取接口实现类的实例。

第二个问题，如果你准备创建一个新的`SecurityService`实现类，它不依赖使用NHibernate程序集的Domain模型，而是使用一个第三方服务总线（比如NServiceBus）来给句柄发送命令消息呢？把这个新的实现加入到Services程序集里仍然会引入对NServiceBus的依赖，不断膨胀的代码库会变得更加脆弱，很难适应后期的新需求。

有个常用的规则就是把接口的实现与接口本身拆分开，分别布置在不同的程序集中。可以使用下节要讲解的阶梯模式来拆分接口和相应的实现类。

2.2.5　阶梯模式

阶梯模式是一种正确组织类和接口的方法。因为接口和接口的实现类布置在不同的程序集内，二者可以独立更改，客户端代码始终只需要引用接口所在的程序集。

你可能会想：“这样做会产生多少程序集啊？如果需要把每个接口和类都拆分到自己独有的程序集内，一个解决方案会不会包含两百个项目啊！”不用担心，因为应用阶梯模式只会增加少量的项目，同时又能让解决方案的结构始终保持清晰明了。如果项目的组织很糟糕，通过应用阶梯模式来减少项目总体数目是有一定效果的。

可以应用阶梯模式再次重构前面的`AccountController`示例，图2-14展示了这次重构的结果。每个实现，也就是每个类，只引用要依赖接口所在的程序集，它不会显式或隐式地引用接口实现所在的程序集。每个实现类也会引用自己接口所在的程序集。阶梯模式的组织方式好处很多：接口没有任何依赖，调用接口的客户端代码也不会有任何隐性依赖，接口的实现也同样只依赖其他仅包含接口的程序集。

这里，我想再详细强调阶梯模式的好处之一：接口不应该有任何外部依赖。开发人员要尽量坚持好这个原则。接口的方法和属性不应当暴露出任何第三方引用中定义的数据对象或类。尽管接口可以（也一定会需要）依赖同一解决方案下的其他项目以及常见的.NET Framework库定义的类，但是应该避免对基础构件实体的依赖。第三方库通常都是用来提供基础构件的。即使使用的是第三方库（比如Log4Net、NHibernate和MongoDB等）的接口，你的接口依然会与库的实现绑定在一起。这是因为这些第三方库的包都应用了随从反模式，而不是阶梯模式。它们都只提供了单个程序集，其中既包含了需要依赖的接口，也包含了不希望依赖的实现。

为了避免这个问题，可以引用用于记录日志、域持久化和文档存储的自定义接口。你的简单接口可以把对第三方的依赖隐藏在对第一方程序集的依赖后面。如果后面需要更换对第三方的依赖，只需要为更换后的新接口编写一个新的适配器就可以。

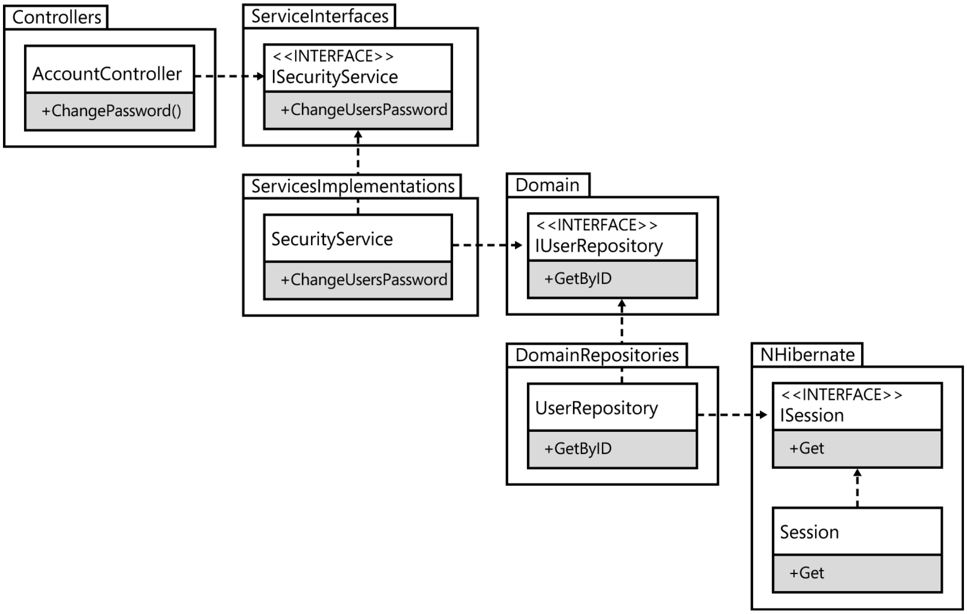


图2-14 阶梯模式的名称恰当地描述了应用该模式后形成的包结构

从实际应用的角度来看，为所有第三方引用都编写适配器和接口并不现实。如果工作量很大，开发团队应当认识到项目不得不保持对第三方程序集的依赖，这种依赖还会慢慢渗透到整个项目。第三方库的规模越大，替换它就越难，耗时也会越长。相比单个巨型第三方库的更好选择应该是框架，后者比前者的规模大得多。

2.2.6 依赖解析

只知道如何组织项目和相关依赖对调试运行时程序集间依赖并没有多大帮助。有时候程序集在运行时并不可用，这就需要找出根本的原因。

1. 程序集

公共语言运行时（Common Language Runtime，CLR）是.NET Framework用来执行代码指令的虚拟机。它也是一个软件产品，作为.NET应用程序的宿主，它以一种可预测的符合逻辑的方式运行。清楚程序集依赖以及修复方案的理论和实践都会很有用。如果知之甚少，在需要诊断程序集问题时可能会走些弯路。

- 解析流程

程序集解析流程在公共语言运行时中很重要。引用程序集或项目后，在运行时加载程序集前就需要解析程序集。这个流程包括几个步骤，期间如果出错，你可以诊断问题可能的原因。

图2-15以流程图的形式展示了程序集解析的过程。这个流程图只是个没有包含所有细节的高层框架，但是用来展示流程的要点已经足够了。流程步骤如下所示。

- 公共语言运行时使用即时（just-in-time，JIT）模型来解析程序集。正如本章开始几节已证实的那样，启动应用时是不会解析应用中包含的引用的，而是直到首次使用程序集的某个特性时才会解析应用中包含的引用（真的是即时解析）。

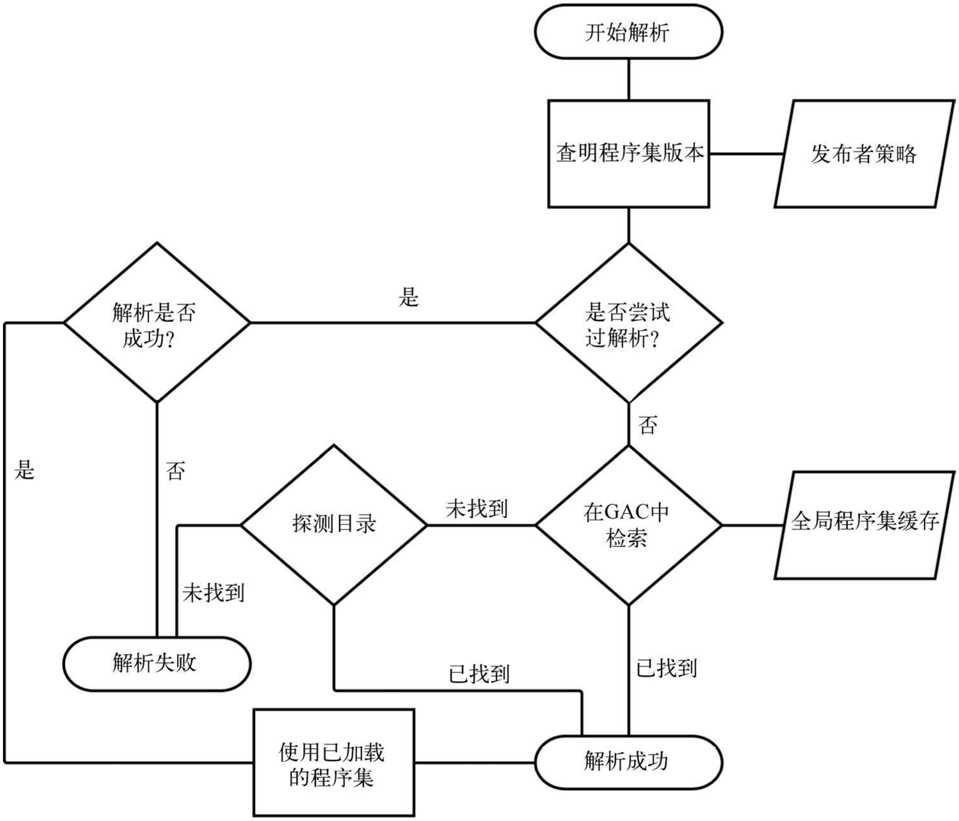


图2-15　程序集解析流程的概要图

- 每个程序集都有一个标识符，它们由程序集名称、版本、文化和公钥令牌组合构成。诸如绑定重定向等特性可以改变程序集的标识符，所以确定标识符也不像看起来那么容易。
- 当程序集标识符确定后，在当前应用程序执行期间，公共语言运行时能够决定是否已经尝试解析过依赖。下面这个Visual Studio项目文件内容片段展示了引用的标识信息。

```
<reference include="MyAssembly, Version=2.1.0.0, Culture=neutral,
   PublicKeyToken=17fac983cbea459c" />
```

- 公共语言运行时会根据是否已经尝试过解析来走不同的分支。如果已经尝试过解析该程序集，那么解析过程要么成功要么失败。如果解析成功了，公共语言运行时会使用已经加载的有效程序集。如果解析失败了，公共语言运行时会知道自己无需再尝试解析了，因为一定会失败。

- 或者，如果是第一次尝试解析该程序集，公共语言运行时会首先检查全局程序集缓存（Global Assembly Cache，GAC）。全局程序集缓存是一个整个机器可见的程序集库，它允许同一个程序集的不同版本在同一个应用程序上执行。如果在全局程序集缓存中找到了该程序集，那么解析过程就成功了，并且会加载找到的程序集。现在你知道了，因为公共语言运行时会首先搜索全局程序集缓存，因此全局程序集缓区中合适的程序集要比程序集文件有更高的优先级。
- 如果在全局程序集缓存中没有找到合适的程序集，公共语言运行时会开始尝试搜索一系列文件夹。可以使用app.config文件中的`codeBase`元素来指定目标文件夹，公共语言运行时会检索这个元素下列出的位置，如果还没有找到合适的程序集，那么后续也不会再去检索其他位置了。此外，默认情况下，公共语言运行时还会去搜索应用的安装根目录，通常情况下是程序的入口文件所在的bin目录。如果在安装目录下也没有找到合适的程序集，解析过程就失败了，公共语言运行时会引发一个异常。典型情况下，应用程序会被终止。

- Fusion日志

Fusion是个很有用的工具，可以用来调试公共语言运行时加载程序集失败的问题。比起尝试使用Visual Studio调试器调试应用程序，更好的办法是打开Fusion日志开关然后查看记录到的日志结果。

要启用Fusion日志，你需要编辑Windows注册表。下面是具体的注册表位置信息。

```
HKLM\Software\Microsoft\Fusion\ForceLog 1
HKLM\Software\Microsoft\Fusion\LogPath C:\FusionLogs
```

其中`ForceLog`值是`DWORD`类型，而`LogPath`是个字符串。你可以将`LogPath`设置为任何你选择的位置。代码清单2-12是个绑定程序集失败的例子。

代码清单2-12　一个尝试绑定程序集失败的Fusion日志示例

```
*** Assembly Binder Log Entry  (6/21/2013 @ 1:50:14 PM) ***

The operation failed.
Bind result: hr = 0x80070002. The system cannot find the file specified.

Assembly manager loaded from:  C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
Running under executable  C:\Program Files\1UPIndustries\Bins\v1.1.0.242\Bins.exe
--- A detailed error log follows.

=== Pre-bind state information ===
LOG: User = DEV\gmclean
LOG: DisplayName = TaskbarDockUI.Xtensions.Bins.resources, Version=1.0.0.0, Culture=en-US,
   PublicKeyToken=null (Fully-specified)
LOG: Appbase = file:///C:/Program Files/1UPIndustries/Bins/v1.1.0.242/
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = Bins.exe
Calling assembly : TaskbarDockUI.Xtensions.Bins, Version=1.0.0.0, Culture=neutral,
```

```
   PublicKeyToken=null.
===
LOG: This bind starts in default load context.
LOG: No application configuration file found.
LOG: Using host configuration file:
LOG: Using machine configuration file from C:\Windows\Microsoft.NET\Framework64
   \v4.0.30319\config\machine.config.
LOG: Policy not being applied to reference at this time (private, custom, partial, or
   location-based assembly bind).
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/en-US/TaskbarDockUI.Xtensions.Bins.resources.DLL.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/en-US/TaskbarDockUI.Xtensions.Bins.resources/
   TaskbarDockUI.Xtensions.Bins.resources.DLL.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/Bins/
   v1.1.0.242/en-US/TaskbarDockUI.Xtensions.Bins.resources.EXE.
LOG: Attempting download of new URL file:///C:/Program Files/1UPIndustries/
   Bins/v1.1.0.242/en-US/TaskbarDockUI.Xtensions.Bins.resources/
   TaskbarDockUI.Xtensions.Bins.resources.EXE.
LOG: All probing URLs attempted and failed.
```

编辑完注册表后，任何托管应用程序的所有解析程序集的尝试（无论成功与否）都会被记录到相应的Fusion日志文件中。显然Fusion下会有大量有用的日志文件产生，但是在大量日志文件中查找问题就像大海捞针一样困难。

幸运的是，Fusion还有个用户界面应用程序，可以帮助开发人员更容易找到自己程序的日志文件，而不用直接在众多的日志文件中苦苦寻找。图2-16展示了Fusion的用户界面。

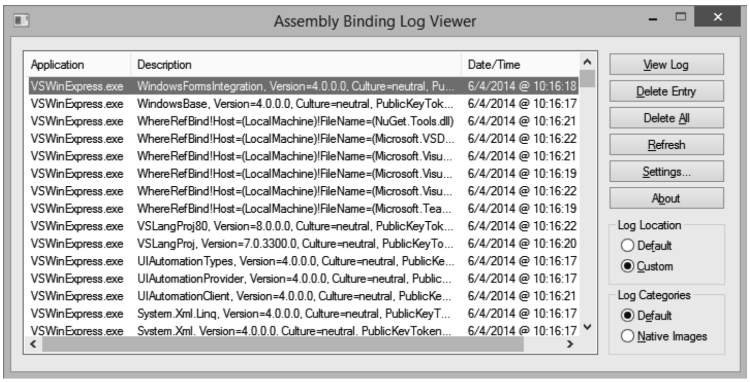


图2-16　Fusion的用户界面可以迅速找到特定程序的日志文件

不是所有的依赖都需要直接引用程序集。一种方式就是将服务代码部署为宿主服务。这种做法需要进程或网络间的数据通讯能力的支持，但是它能最小化客户端和服务之间必需的程序集引用。下一节会详细讲解这一主题。

2. 服务

与程序集相比，客户端和服务之间的耦合关系更加松散，这种方式有利也有弊。根据应用程

序需求的不同，客户端可能很清楚服务的位置，也可能知之甚少。同样，实现服务的方式不多，因此有关服务的需求也不多。选择不同的实现方式，要考虑的取舍不同。

- 已知端点

如果客户端代码编译时就知道服务位置，可以为客户端创建一个服务代理。至少有两种创建代理的方式：使用Visual Studio为项目添加一个服务引用，或者使用.NET Framework的ChannelFactory类编码创建服务代理。

在Visual Studio中为项目添加一个服务引用非常简单：只需要在项目的快捷菜单上选择Add Service Reference（添加服务引用）即可。Add Service Reference对话框只需要知道Web服务定义语言（Web Service Definition Language，WSDL）文件的具体位置，该文件定义了服务的元数据描述、数据类型和可用的行为。选定服务文件后，Visual Studio能够很快为该服务快速生成一组代理类，非常节省时间。Visual Stduio甚至可以生成异步的服务方法以避免出现阻塞。然而，这种方式也有缺点，开发人员对自动生成的代码缺乏控制。如果你自己的编码标准要求比较高，Visual Studio生成的代码可能会不符合要求。另外一个问题是，自动生成的服务代理代码只包含实现类，它不但没有匹配的单元测试，也没有相应的接口定义。

另一个添加服务引用的方式就是通过编码创建服务代理。这种方式最好用于客户端代码能够访问服务接口并且可以通过引用重复使用时。代码清单2-13展示了一个使用ChannelFactory类编码创建服务代理的示例。

代码清单2-13 ChannelFactory能够创建服务代理

```
var binding = new BasicHttpBinding();
var address = new EndpointAddress("http://localhost/MyService");
var channelFactory = new ChannelFactory<IService>(binding, address);
var service = channelFactory.CreateChannel();
service.MyOperation();
service.Close();
channelFactory.Close();
```

ChannelFactory是个泛型类，它的构造函数需要指定服务代理接口。另外，它的构造函数需要传入Binding和EndpointAddress类的对象，因此必须给ChannelFactory类提供完整的地址/绑定/协定（address/binding/contract，ABC）信息。在这个示例中，IService接口就是具体服务实现的接口。ChannelFactory类的CreateChannel方法会返回一个服务代理实例，所有对服务代理实例方法的调用都会调用服务端具体实现中对等的方法。因为是使用同一个服务接口，客户端类型可以通过依赖注入从构造函数传入服务接口参数，这样客户端类型就可以立即变得可测试了。此外，客户端类型也无需知道它们调用的是远程服务。

- 服务发现

有时候，你可能只知道服务的绑定类型或者协定，但并不清楚服务的宿主地址。这种情况下，你可以使用在.NET Framework 4中引入的服务发现特性。

服务发现有两种方式：托管的和自组网的。在托管模式下，有一个被称为发现代理的中

心服务对所有客户端都是公开的，客户端可以直接请求这个中心服务来查找其他可用的服务。这种模式没有多大吸引力，因为它的设计引入了单一故障点（single point of failure，SPOF）的反模式：如果发现代理服务不可用，所有客户端都无法访问其他任何服务，因为它们是不可发现的。

自组网模式不需要发现代理这个中心服务，它采用了组播网络消息的机制。这种模式的默认实现使用了用户数据报协议（User Datagram Protocol，UDP），每个可发现的服务都会在一个特定IP地址[①]和端口上等待查询请求。客户会通过发送组播消息来向网络查询是否有符合查询条件（协议或者绑定类型）的可用服务。举个例子，在自组网模式的场景中，如果一个服务不可用，那么就只有它是不可发现的，而其他的可用服务依然可以响应收到的查询请求。代码清单2-14展示了如何编码托管一个可发现的服务，代码清单2-15则展示了如何通过配置来托管一个可发现的服务。

代码清单2-14　编码托管某个可发现的服务

```
class Program
{
    static void Main(string[] args)
    {
        using (ServiceHost serviceHost = new ServiceHost(typeof(CalculatorService)))
        {
            serviceHost.Description.Behaviors.Add(new ServiceDiscoveryBehavior());

            serviceHost.AddServiceEndpoint(typeof(ICalculator), new BasicHttpBinding(),
   new Uri("http://localhost:8090/CalculatorService"));
            serviceHost.AddServiceEndpoint(new UdpDiscoveryEndpoint());

            serviceHost.Open();
            Console.WriteLine("Discoverable Calculator Service is running...");
            Console.ReadKey();
        }
    }
}
```

代码清单2-15　通过配置托管某个可发现的服务

```
<system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="false"/>
        </behavior>
        <behavior name="calculatorServiceDiscovery">
          <serviceDiscovery />
```

① 这个IP地址是239.255.255.250（IPv4）或[FF02:C]（Ipv6），端口是3702。这是由WS-Discovery标准设置的，无法进行更改配置。

```
          </behavior>
        </serviceBehaviors>
        <endpointBehaviors>
          <behavior name="calculatorHttpEndpointDiscovery">
            <endpointDiscovery enabled="true" />
          </behavior>
        </endpointBehaviors>
      </behaviors>
      <protocolMapping>
          <add binding="basicHttpsBinding" scheme="https" />
      </protocolMapping>
      <serviceHostingEnvironment aspNetCompatibilityEnabled="true"
     multipleSiteBindingsEnabled="true" />
      <services>
        <service name="ConfigDiscoverableService.CalculatorService"
     behaviorConfiguration="calculatorServiceDiscovery">
          <endpoint address="CalculatorService.svc"
     behaviorConfiguration="calculatorHttpEndpointDiscovery"
     contract="ServiceContract.ICalculator" binding="basicHttpBinding" />
          <endpoint kind="udpDiscoveryEndpoint" />
        </service>
      </services>
    </system.serviceModel>
```


要让服务成为可发现的，要做的就是为服务添加ServiceDiscoveryBehavior并托管一个DiscoveryEndpoint。上面两个例子中的UdpDiscoveryEndpoint用来接收客户端发出的组播网络消息。

> **注意** WFC提供的服务发现特性是符合WS-Discovery标准的，因此它也可以与.NET Framework之外的其他不同平台和语言的协议实现互通。

客户端可以使用DiscoveryClient类来查找可发现的服务。首先需要给DiscoveryClient类的实例传入一个DiscoveryEndpoint类的实例；然后创建一个FindCriteria类的实例，该类描述了目标服务的各种属性;最后Find方法使用这个FindCriteria类实例进行查找并返回一个FindResponse类的实例，它的Endpoints属性包含了一组EndpointDiscoveryMetadata实例，其中的每个实例都是一个符合查询条件的服务。代码清单2-16展示了查找可发现服务的这些步骤。

代码清单2-16 服务发现是一种很好的代码解耦方式

```
class Program
{
    private const int a = 11894;
    private const int b = 27834;

    static void Main(string[] args)
    {
```

```
        var foundEndpoints = FindEndpointsByContract<ICalculator>();

        if (!foundEndpoints.Any())
        {
            Console.WriteLine("No endpoints were found.");
        }
        else
        {
            var binding = new BasicHttpBinding();
            var channelFactory = new ChannelFactory<ICalculator>(binding);
            foreach (var endpointAddress in foundEndpoints)
            {
                var service = channelFactory.CreateChannel(endpointAddress);
                var additionResult = service.Add(a, b);
                Console.WriteLine("Service Found: {0}", endpointAddress.Uri);
                Console.WriteLine("{0} + {1} = {2}", a, b, additionResult);
            }
        }

        Console.ReadKey();
    }

    private static IEnumerable<EndpointAddress> FindEndpointsByContract
    <TServiceContract>()
    {
        var discoveryClient = new DiscoveryClient(new UdpDiscoveryEndpoint());
        var findResponse = discoveryClient.Find(new
   FindCriteria(typeof(TServiceContract)));
        return findResponse.Endpoints.Select(metadata => metadata.Address);
    }
}
```

请牢记自组网模式使用的是UDP，而不是TCP，因此无法保证消息投递的结果一定是成功的。可能出现的数据包丢失情况，要么是因为请求包无法到达服务端，要么是因为响应包无法返回客户端。无论是哪种丢包方式，在客户端看来就是处理请求的服务当前不可用。

> **提示**　当使用Internet信息服务（Internet Information Service，IIS）或Windows进程激活服务（Windows Process Activation Service，WAS）托管可发现的服务时，一定要确保使用Microsoft AppFabric AutoStart功能。服务要能被发现，首先服务必须是可用的，这就意味着为了接收客户端的请求，服务必须首先处于运行状态。AppFabric AutoStart特性允许应用程序在IIS中启动时也能自动启动服务。如果没有自动启动，只有在第一次请求该服务时才会启动它。

- REST化服务

创建REST（REpresentational State Transfer，表述性状态转移）化服务的最大好处是客户端几乎没有任何依赖，只需要一个所有语言的框架和库都提供的HTTP client实例。因此REST化服务

非常适合开发需要跨平台的功能强大的服务。举例来说，Facebook和Twitter都为各种查询和命令提供了丰富的REST API。这样能够很容易为各种平台开发客户端，包括Windows Phone 8、iPhone、Android、Windows 8、Linux和其他平台。如果不使用REST，单一的服务端实现很难做到同时支持所有平台上的客户端。

ASP.NET Web API用来创建基于.NET Framework的REST服务。与ASP.NET MVC框架类似，它也允许开发人员创建能直接映射为网络请求的方法。ASP.NET Web API提供了一个名为`ApiController`的基础控制器类，通过继承这个类并实现一些使用HTTP动作（`GET`、`POST`、`PUT`、`DELETE`. `HEAD`、`OPTIONS`和`PATCH`等）作为名称的方法后，当接收到一个带有某个动作名称的HTTP请求时，相应名称的方法就会被执行。代码清单2-17展示了一个实现了所有HTTP动作命名方法的服务。

代码清单2-17 ASP.NET Web API几乎支持所有HTTP动作

```
public class ValuesController : ApiController
{
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    public string Get(int id)
    {
        return "value";
    }

    public void Post([FromBody]string value)
    {
    }

    public void Put(int id, [FromBody]string value)
    {
    }

    public void Head()
    {
    }

    public void Options()
    {
    }

    public void Patch()
    {
    }

    public void Delete(int id)
    {
    }
}
```

代码清单2-18　客户端可以使用`HttpClient`类访问任何REST化服务

```
class Program
{
    static void Main(string[] args)
    {
        string uri = "http://localhost:7617/api/values";

        MakeGetRequest(uri);
        MakePostRequest(uri);

        Console.ReadKey();
    }

    private static async void MakeGetRequest(string uri)
    {
        var restClient = new HttpClient();
        var getRequest = await restClient.GetStringAsync(uri);

        Console.WriteLine(getRequest);
    }

    private static async void MakePostRequest(string uri)
    {
        var restClient = new HttpClient();
        var postRequest = await restClient.PostAsync(uri,
                new StringContent("Data to send to the server"));

            var responseContent = await postRequest.Content.ReadAsStringAsync();
            Console.WriteLine(responseContent);
        }
    }
```

为了强调所有平台上的客户端都几乎一样没有依赖，代码清单2-19展示了一个使用Windows PowerShell 3脚本编码访问服务`GET`和`POST`方法的示例。

代码清单2-19　使用Windows PowerShell 3访问REST化服务一样非常简单

```
$request = [System.Net.WebRequest]::Create("http://localhost:7617/api/values")
$request.Method ="GET"
$request.ContentLength = 0

$response = $request.GetResponse()
$reader = new-object System.IO.StreamReader($response.GetResponseStream())
$responseContent = $reader.ReadToEnd()
Write-Host $responseContent
```

在上面的示例中，脚本代码使用.NET Framework的`WebRequest`类的对象来访问REST化服务。其中，`WebRequest`类是`HttpRequest`的父类，它的`Create`方法是个工厂方法，会根据传入的http://开头的URI字符串返回一个`HttpRequest`类的实例。

2.2.7　使用 NuGet 管理依赖

依赖管理工具能够大大简化依赖管理工作。它们会负责跟踪依赖链并准备好所有要依赖的程序集和相关资料，同时还会负责管理依赖版本。开发人员只需要指定依赖的具体版本，剩下的工作会由管理工具自动完成。

NuGet是一个.NET Framework的包管理工具。这里，NuGet将依赖称为包，其中不仅可以包括程序集，还可以包括配置、脚本以及图像等任何你需要的数据。使用诸如NuGet之类的包管理器的最有说服力的理由之一就是，这些工具对包的依赖关系非常了解。它们能为需要引用包的项目自动导入整个依赖链上所有需要的文件和数据。

从Visual Studio 2013开始，NuGet已经作为默认的包管理工具被完全集成到Visual Studio IDE当中了。

1. 使用包

NuGet为Visual Studio解决方案浏览窗口增添了一些新的快捷菜单项。选择任一菜单，你就可以打开NuGet包管理窗口并添加对依赖的引用。

举个例子，我打算引用Corrugatedlron，它是一个用于存储Riak非Sql键值对的.NET Framework客户驱动程序。图2-17是NuGet包管理窗口的截图。

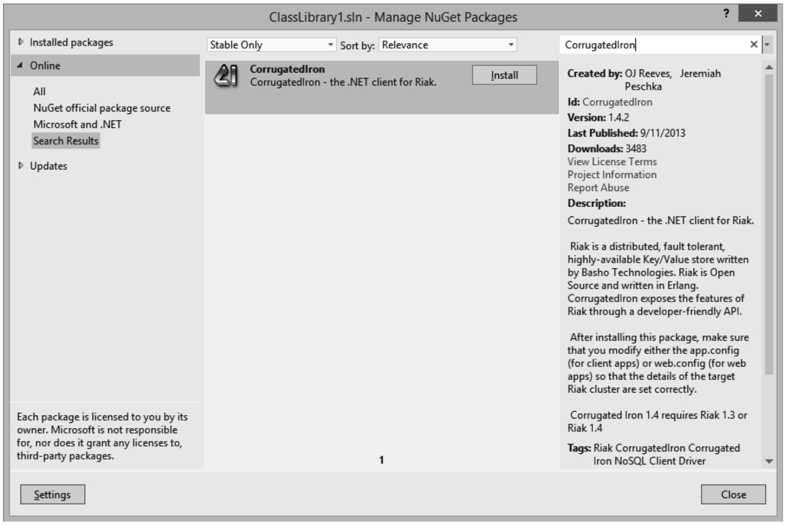


图2-17　NuGet包带有很多有用的元数据

在NuGet包管理窗口的列表中选择了一个包时，右侧的信息面板就会显示出这个包的一些元数据，其中包括：独一无二的命名、作者、版本、最后修改日期、描述以及自身的所有依赖。在引用一个包前，首先要根据版本要求安装并引用这个包自身的所有依赖。以Corrugatedlron为例，它需要一个版本不低于4.5.10的Newtonsoft.Json包、一个.NET Framework JSON/类序列化器和一个版本不低于2.0.0.602的protobuf-net包。而这些被Corrugatedlron依赖的包自身也都有一些依赖。

当你选择安装包后，NuGet首先会尝试下载所有相关文件并把它们存放在解决方案的packages/文件夹下。这样做的好处是，与本章开始介绍的手动创建dependencies/文件夹一样，也可以对整个文件夹进行源代码管理。当你需要使用这些库时，NuGet会自动将下载好的程序集加入到引用列表中。图2-18展示了添加Riak包后的项目引用列表。

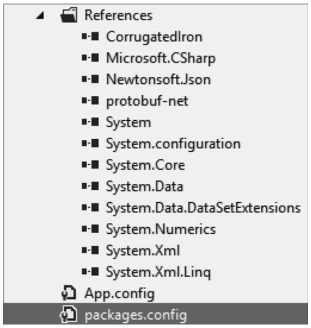


图2-18　NuGet工具会将目标包及其所有依赖一起自动加入到了项目的引用列表中

除了添加对包的引用外，NuGet工具还会创建一个包含项目所引用包及其版本信息的packages.config文件。这些信息会在NuGet工具升级和卸载包时用到。

在真正使用Riak的功能前，还需要进行一些默认的配置。所以NuGet不只是为项目下载和引用了很多程序集，它还自动根据Riak功能的需要在项目的app.config文件中设置了一些默认值。代码清单2-20展示了安装Riak之后的app.config文件内容。

代码清单2-20　NuGet工具在app.config文件中专门为Riak添加了一个新的`configSection`

```
<configuration>
  <configSections>
    <section name="riakConfig" type="CorrugatedIron.Config.RiakClusterConfiguration,
   CorrugatedIron" />
  </configSections>
  <riakConfig nodePollTime="5000" defaultRetryWaitTime="200" defaultRetryCount="3">
    <nodes>
      <node name="dev1" hostAddress="riak-test" pbcPort="10017" restScheme="http"
restPort="10018" poolSize="20" />
      <node name="dev2" hostAddress="riak-test" pbcPort="10027" restScheme="http"
restPort="10028" poolSize="20" />
      <node name="dev3" hostAddress="riak-test" pbcPort="10037" restScheme="http"
restPort="10038" poolSize="20" />
      <node name="dev4" hostAddress="riak-test" pbcPort="10047" restScheme="http"
restPort="10048" poolSize="20" />
    </nodes>
  </riakConfig>
</configuration>
```

很显然，使用NuGet 工具能为你节省大量的时间。你无需花费精力在Riak的官网上下载Corrugatedlron及其所有依赖程序集。这有助于你专注于真正的开发工作上。当需要将Corruga-

tedlron升级到新版本时，你只需要使用NuGet工具自动为整个解决方案更新所有相关的包即可。

2. 制作包

NuGet工具还提供了创建新包的功能。你可以自己创建包并将其发布到NuGet的官方商店，这样其他开发人员就可以使用你的自制包，或者你想把所有第一方依赖制作成包以便在项目内部同步引用。图2-19展示了使用NuGet包浏览器创建自制包的截图。在我的这个自制包里，我把CorrugatedIron设置为依赖项，因此它也会隐式依赖Newtonsoft.Json和Protobuf-net这两个包。我还为这个包添加了一个专门针对.NET Framework4.5.1的库工件，还添加了一个将在My folder/NewFile.txt下引用的程序集中创建的文本文件。包含NewFile.txt的文件夹MyFolder，它们会被复制到包的安装目录下。此外，还有一个Windows PowerShell脚本，它会在包的安装过程中执行。可以在这个脚本中完成很多自定义动作，在此需要感谢一下强大的Windows PowerShell。

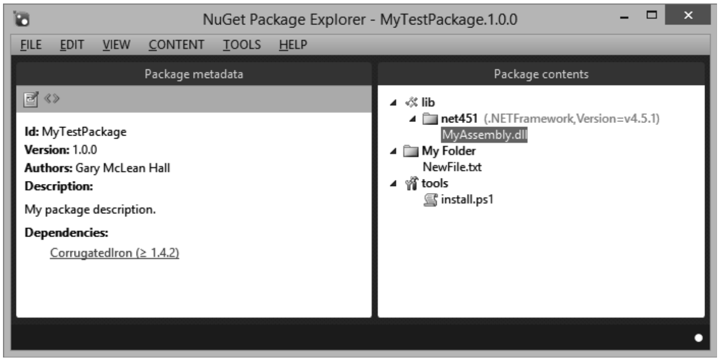


图2-19 使用NuGet包浏览器可以轻松创建自己的包

NuGet生成的每个包都会有一个XML文件，其中包含了要在安装窗口中显示的详细元数据。代码清单2-21展示了XML文件内容的一个示例。

代码清单2-21 包含了包的详细元数据的XML定义

```
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
    <metadata>
        <id>MyTestPackage</id>
        <version>1.0.0</version>
        <authors>Gary McLean Hall</authors>
        <requireLicenseAcceptance>false</requireLicenseAcceptance>
        <description>My package description.</description>
        <dependencies>
            <dependency id="CorrugatedIron" version="1.0.1" />
        </dependencies>
    </metadata>
</package>
```

对于一直痛苦地手动打理大量第三方依赖的人来说，NuGet这个高效的工具真可谓是个天大的福利。而实际上，NuGet也并不局限于管理第三方依赖。当一个解决方案的规模变得足够大时，

最好是能通过分层将整个解决方案划分为多个部分。可以把每一层的所有程序集放入一个NuGet包以供上层使用。这样划分后得到的多个小规模的解决方案很容易进行协调管理。

3. 工具Chocolatey

与NuGet工具类似，Chocolatey也是一种包管理工具。不同的地方是，NuGet的包是一些程序集，而Chocolatey的包是一些应用程序和工具。了解Linux的开发人员会发现Chocolatey有点像Debian和Ubuntu系统上的包管理器`apt-get`。再啰嗦一遍，包管理工具的好处有：简易安装、依赖管理以及轻松使用。

下面的Windows PowerShell脚本用来下载和安装Chocolatey。

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object
   net.webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET
   PATH=%PATH%;%systemdrive%\chocolatey\bin
```

安装好Chocolatey后，你可以使用命令行搜索和安装各种应用和工具。Chocolatey的安装程序已经更新了命令行路径，以包含Chocolatey.exe应用。Chocolatey和Git一样也有一些诸如`list`和`install`之类的子命令，不同的是，它还分别为这些子命令提供了诸如`clist`和`cinst`之类的快捷方式。代码清单2-22展示了一个Chocolatey会话示例，用来搜索和安装名为FileZilla（一个FTP客户端应用程序）的包。

代码清单2-22　查找并安装需要的应用程序包

```
C:\dev> clist filezilla
ferventcoder.chocolatey.utilities 1.0.20130622
filezilla 3.7.1
filezilla.commandline 3.7.1
filezilla.server 0.9.41.20120523
jivkok.tools 1.1.0.2
kareemsultan.developer.toolkit 1.4
UAdevelopers.utils 1.9
C:\dev> cinst filezilla
Chocolatey (v0.9.8.20) is installing filezilla and dependencies. By installing you accept the
license for filezilla and each dependency you are installing.
 . . .
 This Finished installing 'filezilla' and dependencies - if errors not shown in console, none
detected. Check log for errors if unsure.
```

只要Chocolatey没有报错，请求的包就已经成功安装了。有一点要警惕的是，Chocolatey为了从命令行执行新安装应用程序或工具的二进制文件，可能会修改系统`PATH`环境变量。Chocolatey工具能够搜索到大量的应用程序和工具包，这就是它的强大优势。

2.3　分层

至此，本章前面都是在讲解程序集层次的依赖管理。当然，管理好程序集层次的依赖很自然是组织应用程序的第一步，因为所有类和接口都包含在程序集中，而且这些程序集之间如何关联

也是开发人员普遍担心的问题。组织好程序集层次的依赖关系后，所有程序集会包含附属于一组相关功能的类和接口。然而，你又如何保证程序集的组划分是正确的呢？

在开发的软件系统中，几个相关的程序集会形成一个组件。以相似且定义良好的结构化方式管理组件间的互动和管理程序集层次上的依赖关系一样重要（甚至更重要）。如图2-20所示，一个组件通常不是要最终部署的DLL或EXE文件，而是一个逻辑程序集组，组中的这些程序集在功能上是相关的。

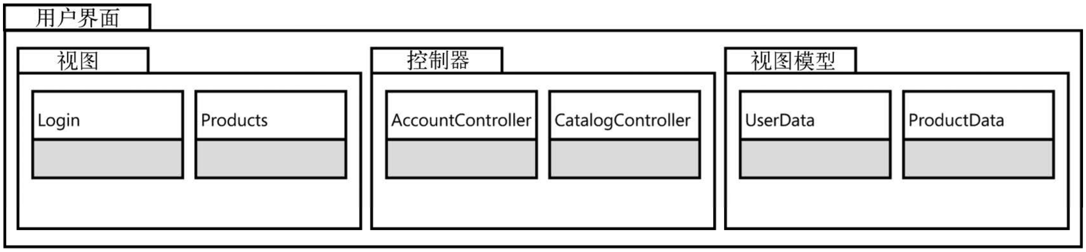


图2-20　可以通过将功能相关的程序集划分到同一个组来定义逻辑组件

图中包括了三个程序集：视图、控制器和视图模型。每个程序集包含了两个类型，它们的功能不同，需要的依赖也可能不同。图2-20中用户界面包所包含的类型和程序集都是逻辑上的概念，而不是实际的.NET Framework类和程序集实体。你可以把这三个程序集放置在解决方案下的一个名为UserInterfaces的文件夹下，也可以再将一个与用户界面没有任何关系的程序集加入进来，当然，这会让用户界面这个分组变得名不副实。你自己要多多留意，因为没有其他办法能防止出现这种情况。

在依赖管理的上下文中，组件和其他更低层次的编码概念没有差别。与方法、类以及程序集一样，*层次*（layer）也可以作为本章前面所讲的依赖图中的节点。因此，对层次节点也可以应用相同的规则：确保有向图无环且提供单一职责。

分层（layering）是一种架构模式，它鼓励开发人员将软件组件看作是水平功能层，而一个完整的应用程序可以划分为多个水平功能层。分层形成的组件一个叠加在另外一个上面，它们的依赖关系方向必须朝下。也就是说，程序最底层的组件没有依赖[①]，每个层只能依赖它的直接下层。通常情况下，应用程序的顶层都是用户界面，服务程序的顶层都是客户端用来与服务端交互的API。

2.3.1　常见的模式

任何项目都可以从常见的几个分层模式中找到适合自身的模式。本节要介绍的几个分层模式只是仅供参考，你还需要对它们进行定制以符合实际项目的具体需求和限制。这几种分层模式之间唯一的区别就是分层数目不同。本节会先介绍最简单的两层划分模式，然后讲解加入中间层的

① 严格来讲，还可能有对第三方的基础构件程序集的依赖，“没有依赖”在这里只是为了表示最低层不再依赖任何第一方的代码。

三层划分模式，最后引入任意分层模式的介绍。

需要的分层数目与方案的复杂度相关，而方案的复杂度又与问题的复杂度相关。因此，问题越复杂，越可能引入更多分层的架构。在这种情况下，复杂度由很多因素决定，其中包括：项目的时间限制、需要的持久度、需求的变更频率以及开发团队对模式及其实践的重视程度等。

因为本书旨在讲解如何适应需求变更，因此，我主张尽量从最简单的方案开始，后面有需要时才将它重构为更复杂的方案。这种方式对项目开发有很多好处。能尽快交付一些成果给客户并且能够尽早获得大量的重要反馈。总是追求很完美的方案是没有意义的，因为客户心中的完美与开发团队想象的完美有可能不一样。多层架构要比简单的两层划分方案耗费更多的开发时间，也无法及时获取重要的用户反馈。

逻辑层与物理层

逻辑层和物理层之间的区别就是代码逻辑组织和物理部署的区别。逻辑上，你可以把应用程序的代码拆分为多个**逻辑层**（layer），但是物理上，你依然可以把它们部署在同一个**物理层**（tier）上。物理层的数目就是单个应用程序拆分部署的宿主机器数目。如果整个应用程序被部署在同一台机器上，也就是说应用被部署在单个物理层上了。如果应用程序（至少有两个逻辑层）被拆分部署在两台独立的机器上，也就是说应用被部署在两个物理层上了。

采用多物理层的部署方式，就意味着不同物理层上同一应用程序的不同逻辑层间的交互会跨越网络边界，这自然也会产生时间性能上的相应代价。同一台机器上的跨进程交互的时间代价已经比较高了，而跨越网络边界交互的时间代价比前者还要高出很多。尽管如此，多物理层的部署方式依然有一个明显的优势，那就是它赋予应用程序更好的扩展能力。假设有一个三层（包括用户界面层、逻辑层和数据访问层）划分架构的网络应用程序，它被部署在单台机器上（也就是单个物理层上），那么这台机器能够支持的用户数目一定不高，因为它本身需要完成所有三个逻辑层的众多任务。如果将应用程序拆分部署在两个物理层上（把用户界面和逻辑层部署在一台机器上，而把数据访问层部署在另外一台独立的机器上），你不仅可以横向扩展用户接口逻辑层，还可以纵向扩展它。

为了纵向扩展，你只需要通过添加内存和处理单元等方式增加机器的能力，因为机器的能力增强了，本身就能完成更多的任务了。此外，你也可以通过增加执行相同任务的新的独立机器来实现横向扩展。这样，会有多台机器托管同样的网页用户界面代码，负载均衡器会实时将客户端的请求分配给最空闲的机器处理。当然，这种部署方式并不能解决网络应用场景中的多用户并发访问的问题。因为同一用户的多个不同请求可能由不同的机器来处理，这需要你谨慎处理好数据缓存和用户身份验证。[①]

1. 两层划分

对没有明确分层方案的最简单改进就是两层划分的方案。尽管两层方案的应用场景也不多，

① 后面当逻辑层和物理层同时出现时才会使用全称，否则layer和tier都会简称为层。——译者注

但是实现它所需的时间非常短。这两个层次分别是用户界面层和数据访问层。但是请记住，只有两个层次并不是只有两个程序集，而是有两组逻辑相关的程序集：一组与用户界面直接相关，另外一组与数据访问相关。

图2-21的UML图展示了两个层次的依赖关系，每个层次都包含了一组逻辑相关的程序集。无论分层数目的多少，每个层次都必须且只能依赖直接下层。

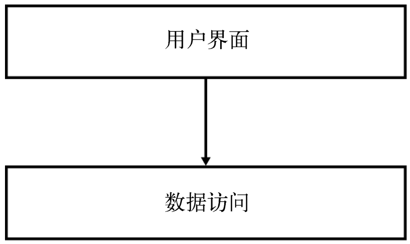


图2-21　一个两层划分的应用由界面组件和数据访问组件组成

- *用户界面层*

用户界面层的职责包括以下四项。

- 为用户提供与应用程序交互的方式（比如：桌面窗口和控件、网页或者带有命令行或菜单的控制台应用程序）。
- 向用户展示数据和信息。
- 接收用户的查询或命令请求。
- 验证用户的输入。

用户界面层有很多种不同的实现方式。它可以是一个带有绚丽图像和动画的WPF客户端，也可以是一组导航网页，抑或是一个带有命令行开关参数或简单菜单（供用户选择以及执行查询或命令）的控制台应用程序。

> **注意**　有些情况下，用户界面会被一组为客户端提供功能的服务代替。这组服务并不是真正可视的用户界面，但是两层划分的架构依然很清晰，只是用服务API层代替了用户界面层。

用户界面层可以使用数据访问层的功能，然而，正如本章前面所讲的，用户界面层不应该直接引用数据访问层具体实现所在的程序集。这两个层次的接口和实现程序集也应该是严格分开的。图2-21的分层经过改进后看起来如图2-22所示。

实际上，这就是在通过阶梯模式解决随从反模式带来的缺陷，只是这里的缺陷是架构级别上的问题。每个层次都是由上层所需功能的抽象以及该抽象的实现组合而成。如果一个层次开始引用直接下层的部分实现，那么这个下层被称为*抽象漏洞*（leaky abstraction）。因为对该层实现的依赖会逐渐蔓延到更高的上层中，从而引入了本来可以避免的依赖关系。

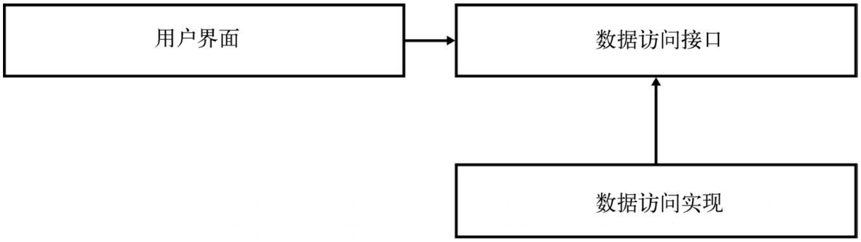


图2-22　两个层次会各自分割为各自相关的实现程序集和接口程序集

● 数据访问层

数据访问层的职责包括以下两项。

- ❑ 响应数据查询请求。
- ❑ 序列化对象模型到关系模型，反序列化关系模型到对象模型。

与用户界面层一样，数据访问层的实现也可以有很多种。这一层通常会包含某种持久数据存储器，它可能是诸如SQL Server、Oracle、MySQL或PostgreSQL等关系型数据库，也可以是诸如MongoDB、RavenDB或Riak等文档型数据库。除了数据存储机制之外，还可能存在一个或多个辅助程序集。这些辅助程序集要么通过调用存储过程来执行查询或插入/更新/删除命令，要么通过Entity Framework或NHibernate将数据映射到关系型数据库。

数据访问层的所有接口都应该隐藏所有与技术相关的事情，也不应该引入任何对第三方的依赖，这样才可以保证客户端完全不受具体实现选择的影响。

设计良好的数据应用层能够在多个应用程序中重用。如果两个用户界面需要把相同的数据展示为不同的表格形式时，它们就可以共享同一个数据访问层。假设一个应用程序需要同时支持Windows 8和Windows Phone 8，虽然两个平台上的用户界面需求不同，但是都可以使用同样的数据访问层。

与其他架构方式一样，在实际采用两层划分之前需要清楚该方案的优缺点。以下是适合采用两层划分架构的一些场景。

- ❑ 应用程序只有一些琐碎的数据验证且没有多少业务逻辑，可以将它们直接归到数据访问层或用户界面层中。
- ❑ 应用程序主要执行数据的创建、读取、更新和删除（creating, reading, updating, and deleting，CRUD）操作。在用户界面和数据访问层间增加额外层会导致CRUD变得更加困难。
- ❑ 时间太仓促。如果只是需要开发一个原型或模拟程序，限制分层数目会节省很多开发时间，也能让概念验证的可行性更加明确。如果你能坚持用好诸如阶梯模式等好的开发实践，后续需要额外的层次时再添加也会更容易。

但是，两层架构也有明显的缺陷，它不适合在以下场景中应用。

- ❑ 应用程序预期或已确定会有复杂的业务逻辑。从技术角度讲，将业务逻辑放入用户界面层或数据访问层会破坏这两层的设计初衷，导致它们变得不够灵活且难以维护。
- ❑ 应用程序已明确在一两个冲刺后会需要多于两层的架构。如果一个临时架构只维持短短

的几周时间，那么为了能尽快得到反馈而快速实现这个临时方案其实是不值得的。

两层架构依然是一个很实用的选择，但很多开发人员都会着迷于最新的架构趋势而忽视这些简洁的设计，他们会把一个简单的应用复杂化，不仅错失及时的用户反馈，而且难以维护。通常情况下，最简单的方案可能就是正确的方案。

2. 三层划分

三层划分的架构是在用户界面层和数据访问层之间增加了一个业务逻辑层。应用会在增加的业务逻辑层中封装更复杂的处理逻辑。与数据访问层一样，同一份业务逻辑层也可以由不同的用户界面层重用。图2-23是一种典型的三层架构。

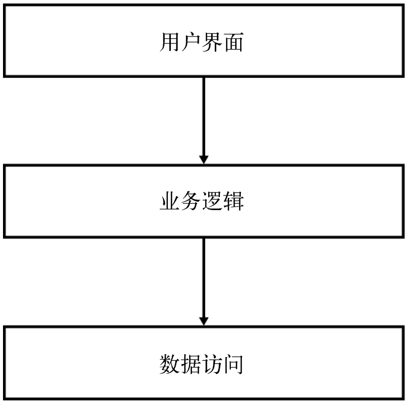


图2-23　中间层包含了应用的处理或业务逻辑

再强调一次，业务逻辑层和数据访问层一样，需要为客户端提供接口和实现两个程序集，要避免成为抽象漏洞。

> **注意**　尽管网络应用大多数都采用三个逻辑层的架构方式，但是通常是部署在两个物理层上的。一个物理层专门负责数据库，另外一个物理层负责其余事情：用户界面、业务逻辑甚至部分数据访问工作。

- 业务逻辑层

业务逻辑层的职责包括以下两项。

- ❑ 处理来自用户界面层的命令。
- ❑ 为业务域的流程、规则和工作流建模。

业务逻辑层有可能是一个命令处理器，它会在接收用户通过用户界面层下达的命令后，协同数据访问层一起解决某个具体问题或执行某项特别任务。业务逻辑层也可以是一个业务域模型，它把整个业务的所有过程映射在软件设计和实现当中。对于后者，通常会在数据访问层中增加一个对象/关系映射（Object/Relational Mapping，ORM）组件，这样可以通过域驱动设计（domain-driven design，DDD）的方式直接实现逻辑层的类型。域模型应该没有任何依赖，既不

依赖任何下层，也不依赖具体的实现技术。举个例子，域模型程序集不应该依赖对象/关系映射库，而应该创建一个独立的映射程序集，它包含了如何指导对象/关系映射库映射到域模型的具体实现。这样做，就可以重用那些不依赖对象关系映射库的域模型核心类型，更换对象关系映射库也不会影响域模型和客户端代码。图2-24展示了带有域模型的逻辑层的一个可能的实现。

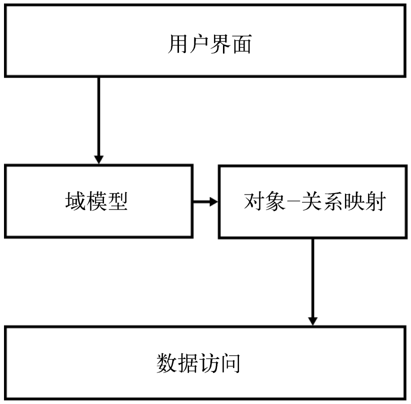


图2-24　域模型的程序集如何协作来形成逻辑层

如果应用的逻辑比较复杂，比如是那些会影响人们真实工作流的业务规则，就有必要为此增加一个逻辑层。另外，即使逻辑不是特别复杂但变更却很频繁，也应该引入逻辑层来封装这部分逻辑。增加的逻辑层能够简化用户界面层以及数据访问层的实现，以便它们集中完成自身的本职工作。

2.3.2　纵切关注点

有时候，一个组件的职责很难集中在单个层次内。诸如审核、安全以及缓存等功能在应用程序的每个逻辑层都有可能存在。如果无法使用Visual Studio调试器单步调试那些已经部署到终端机器上的应用代码，可以使用日志手动追踪每个方法在调用和返回点处的代码行为来辅助调试。代码清单2-23展示了一个记录方法的传入参数值和返回值的示例。

代码清单2-23　手动记录纵切关注点可以很快了解到代码的意图

```
public void OpenNewAccount(Guid ownerID, string accountName, decimal openingBalance)
{
    log.WriteInfo("Creating new account for owner {0} with name '{1}' and an opening
   balance of {2}", ownerID, accountName, openingBalance);

    using(var transaction = session.BeginTransaction())
    {
        var user = userRepository.GetByID(ownerID);
        user.CreateAccount(accountName);
        var account = user.FindAccount(accountName);
        account.SetBalance(openingBalance);

        transaction.Commit();
```

```
    }
}
```

示例中的日志方式耗时费力且容易出错，每个方法都会出现这些与方法主题无关的代码，从而导致有效代码率降低。更好的方式应该是把这些提取出的纵切关注点进行封装并以一种更优雅的方式应用到源代码中。面向切面编程（Aspect-Oriented Programming，AOP）是一种很常见的增加功能的优雅方式。

1. 切面

面向切面编程是代码中跨层次的纵切关注点（也称为切面）的运用。.NET Framework有多个面向切面编程库以供选择（可以使用NuGet搜索AOP），但是下面的示例使用的是PostSharp，它有个免费但受限的版本可用。代码清单2-24展示了如何使用PostSharp定义的扩展属性追踪代码行为。

代码清单2-24 切面是个实现纵切关注点的好方式

```
[Logged]
[Transactional]
public void OpenNewAccount(Guid ownerID, string accountName, decimal openingBalance)
{
    var user = userRepository.GetByID(ownerID);
    user.CreateAccount(accountName);
    var account = user.FindAccount(accountName);
    account.SetBalance(openingBalance);
}
```

附加在`OpenNewAccount`方法上的两个扩展属性提供了与代码清单2-23一样的功能，但是明显更优雅简洁。`Logged`属性能够将方法调用及其参数值记录到日志文件中。`Transactional`属性实现了数据库事务处理功能，如果动作成功就提交事务，如果失败就会回滚事务。这两个属性的最大优势就是它们能够应用于任何方法，而不仅仅局限于示例中的这个方法，因此这种属性可以在代码中大量重用。

2.3.3 非对称分层

所有用户的请求都是通过应用的用户界面传达的，然而，收到请求后的处理过程不一定完全相同。取决于请求类型，分层也可以是非对称的。恰当的分层要考虑是否对于处理有些请求太过复杂或不足，还要考虑是否实用。

最近几年，命令/查询职责分离（Command/Query Responsibility Segregation，CQRS）这种非对称分层模式变得很流行。下面在讲解命令/查询职责分离这个架构模式之前，需要先讨论一个方法层次的基础原则：命令/查询分离（Command/Query Separation，CQS)。

1. 命令/查询分离

命令/查询分离是Bertrand Meyer在其著作*Object-Oriented Software Construction*（1997年由Prentice Hall出版）中首次提出的一个方法层次的原则，它认为任一对象方法要么是命令，要么

是查询。

命令是对动作的强制调用，需要代码做某些动作。这种命令方法可以改变某些系统状态但不应返回值。代码清单2-25展示了两个命令方法，第一个符合命令/查询分离原则，第二个则不符合。

代码清单2-25　一个符合命令/查询分离原则的命令方法和另一个不符合命令/查询分离原则的命令方法

```
// Compliant command
Public void SaveUser(string name)
{
    session.Save(new User(name));
}
// Non-compliant command
public User SaveUser(string name)
{
    var user = new User(name);
    session.Save(user);
    return user;
}
```

查询是对数据的请求，需要代码获取某些数据。这种查询方法为客户端代码返回数据但不应改变任何系统状态。代码清单2-26展示了两个查询方法，第一个符合命令/查询分离原则，第二个则不符合。

代码清单2-26　一个符合命令/查询分离原则的查询方法和另一个不符合命令/查询分离原则的查询方法

```
// Compliant query
Public IEnumerable<User> FindUserByID(Guid userID)
{
    return session.Get<User>(userID);
}
// Non-compliant query
public IEnumerable<User> FindUserByID(Guid userID)
{
    var user = session.Get<User>(userID);
    user.LastAccessed = DateTime.Now;
    return user;
}
```

命令方法和查询方法签名上的唯一区别就是有无返回值。如果一个符合命令/查询分离原则的方法返回一个值，那么你就可以大胆假设该方法不会改变任何系统对象状态。这样做带来的一个优势就是，你可以任意调整查询方法的顺序，因为它们对对象状态没有任何影响。如果一个符合命令/查询分离原则的方法没有返回值，你就可以认为它改变了对象的状态。对于命令方法，你需要留意不要随意改变它们的调用顺序。

2. 命令/查询职责分离

命令/查询职责分离模式是由Greg Young首先提出的。命令/查询职责分离模式是方法层次上

的命令/查询分离原则在架构层上的应用，也是一种常见的非对称分层模式。基于命令/查询分离原则，命令/查询职责分离模式提出：命令和查询可能需要以不同的路径通过不同的逻辑层达到最优处理的效果。

举个例子，带有域模型的三层架构可以应用最简单的命令/查询职责分离模式。此时，只有来自应用程序的命令才会使用域模型，而来自应用程序的查询则会跳过逻辑层。图2-25展示了这个设计。

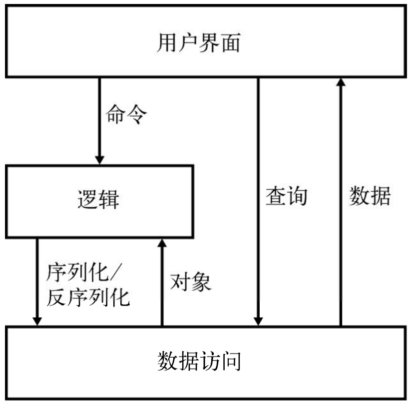


图2-25　域模型仅应该用于处理来自上层的命令

查询数据通常需要足够快，也不保证具有事务一致性：为了及时响应，不完整或混乱的数据读取是可以接受的。不同的是，命令处理通常都需要保证具有事务一致性，因此由不同的层次处理命令和查询是有意义的。有些时候，数据访问层也可以区分命令和查询。由完全符合ACID标准（ACID是atomic, consistent, isolated, durable的缩写，它们的意思分别是原子的、一致的、可隔离的和持久的）的数据库处理命令，而简单的文档存储对查询来说就已经够用了。为了保证查询结果最终是一致的，可以由命令层触发事件来异步更新文档存储。

2.4　总结

本章已经展示了开发软件时依赖组织可能导致的严重问题。合理的依赖管理可以为项目带来长期的健康、良好的自适应和生存能力。如果开发人员鲁莽地创建了相互引用的类型，必然会引入乱作一团的依赖关系，这会严重影响团队持续地定期交付业务价值的能力。

从互有交互的独立方法和类型，到程序集引用，再到架构层的组件划分，所有层次上的依赖关系都需要认真管理。开发人员必须要时刻警惕那些从自己的方法、类型、程序集或层上泄漏的错误依赖。

在某种程度上，本章是本书很多剩余内容的基础。后续所有章节还会不断地讲解如何编写可维护的自适应代码。当然，如果程序集的引用是一团乱麻，层接口对外暴露了最底层的依赖，那么代码就一定会变得很难测试、修改和理解，而此时才去尝试使用任何模式或最佳实践则都已为时过晚。

第 3 章

接口和设计模式

完成本章学习之后，你将学到以下技能。

- 定义接口并识别接口与类的主要区别。
- 在接口中应用诸如适配器和策略等设计模式。
- 通过鸭子类型、混合类型和流接口等了解接口的多样性。
- 识别接口的局限并实现变通方案。
- 识别接口常见的反模式和过度使用的情况。

接口是Microsoft .NET Framework开发中一个非常强大的构件。尽管关键字`interface`很简单，但是它代表了一个非常强大的范式。如果正确应用，接口定义的扩展点会让你的代码具有非常好的适应变更的能力。然而，有些不好的接口应用方式还是很常见的。

本章会给出接口和类之间的区别，还会讲解如何让二者协同发挥最优作用：不但能够防止实现的变更影响客户端代码，还可以充分利用多态的能力。

本章也会讲述接口的多样性，它是现代软件方案中一个无处不在的强大工具。有一些基于接口的强大设计模式，如果能够正确应用（配合本书中讲述的其他模式），就可以让代码变得非常灵活，也可以很好地适应敏捷项目所拥抱的需求变更。

然而，单单使用接口并不能解决所有问题。只有以正确的方式，谨慎且适量地使用接口才能为项目带来好处。本章会提到一些经常被滥用的接口应用方式。

3.1 接口是什么

接口定义了类的行为，但并不定义如何实现行为。接口和类是不同的概念，但是接口需要一个类实现接口所定义的行为。

从开发语言角度来看，语法关键字`interface`定义了接口，这个简单的关键字代表了接口需要的一切。从非开发语言的角度来看，接口也由它们所包含的特性定义，特性是指接口要表达和启用的概念。

3.1.1 语法

C#使用关键字`interface`来定义接口。与类一样，接口可以包括属性、方法和事件。然而，

接口中的任何元素都不需要设定访问权限，因为它们默认都是公开的，实现接口的类必须以public方式实现接口的所有元素。代码清单3-1展示了一个接口的声明和可能实现。

代码清单3-1 一个接口的声明和实现

```
public interface ISimpleInterface
{
    void ThisMethodRequiresImplementation();

    string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get;
    }

    public event EventHandler<EventArgs> InterfacesCanContainEventsToo;
}
// . . .
public class SimpleInterfaceImplementation : ISimpleInterface
{

    public void ThisMethodRequiresImplementation()
    {

    }

    public string ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
    }

    public int ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
             return this.encapsulatedInteger;
        }
        set
        {
            this.encapsulatedInteger = value;
        }
    }

    event EventHandler<EventArgs> InterfacesCanContainEventsToo = delegate { };

    private int encapsulatedInteger;
}
```

.NET Framework不支持继承多个基类的概念，但它支持一个类同时实现多个接口。

理论上，一个类能实现的接口数目并没有限制，只要根据项目实际情况决定一个上限即可。代码清单3-2在前面示例基础上增加了一个接口。

代码清单3-2　一个类可以实现多个接口

```
public interface IInterfaceOne
{
    void MethodOne();
}
// . . .
public interface IInterfaceTwo
{
    void MethodTwo();
}
// . . .
public class ImplementingMultipleInterfaces : IInterfaceOne, IInterfaceTwo
{
    public void MethodOne()
    {
    }

    public void MethodTwo()
    {
    }
}
```

一个类可以实现多个接口，同样，多个类也可以实现同一个接口。

多重继承

有些编程语言，特别是C++，支持继承多个基类的概念。然而，.NET Framework语言不允许这种特性，如果发现有类尝试继承两个或更多基类，编译器会给出如图3-1所示的警告。

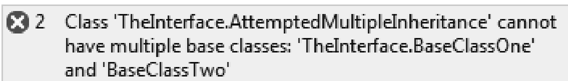


图3-1　编译器会阻止多重继承

钻石继承问题

.NET Framework不支持多重继承的原因之一就是它会引起钻石继承问题。当一个类继承于两个或更多包含相同方法的基类时，这个派生类应该使用哪个方法呢？图3-2展示了钻石问题的UML图。

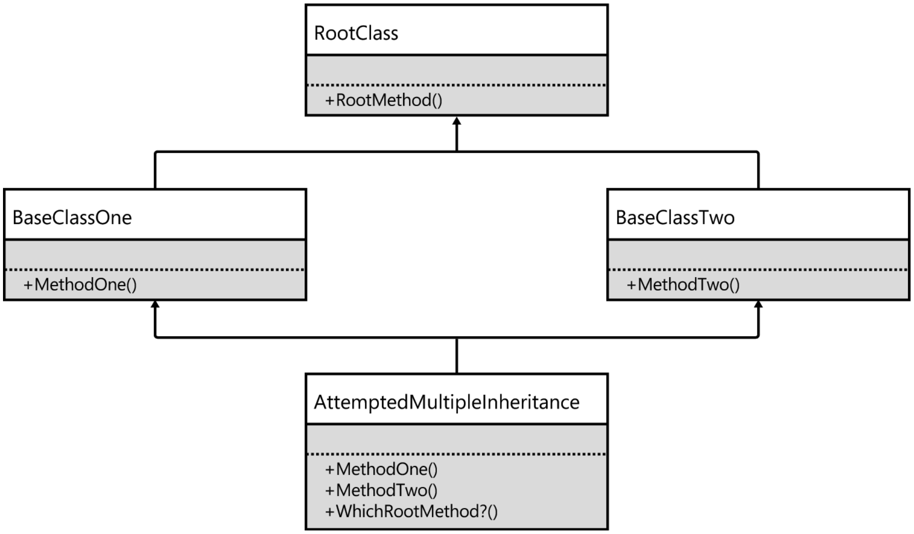


图3-2 一个展示了钻石继承问题的UML图

这种情况下，AttemptedMultipleInheritance类应该继承哪个基类的RootMethod方法，是BaseClassOne类的还是BaseClassTwo类的？正是由于存在这种歧义性，.NET Framework才决定不支持类的多重继承。

3.1.2 显式实现

接口也可以显式实现。显式实现与上一节示例展示的隐式实现是有区别的。代码清单3-3展示了与前面示例相同的类，但它是显式实现了接口。

代码清单3-3 显式实现接口

```
public class ExplicitInterfaceImplementation : ISimpleInterface
{
    public ExplicitInterfaceImplementation()
    {
        this.encapsulatedInteger = 4;
    }

    void ISimpleInterface.ThisMethodRequiresImplementation()
    {
        encapsulatedEvent(this, EventArgs.Empty);
    }

    string ISimpleInterface.ThisStringPropertyNeedsImplementingToo
    {
        get;
        set;
```

```
    }

    int ISimpleInterface.ThisIntegerPropertyOnlyNeedsAGetter
    {
        get
        {
            return encapsulatedInteger;
        }
    }

    event EventHandler<EventArgs> ISimpleInterface.InterfacesCanContainEventsToo
    {
        add { encapsulatedEvent += value; }
        remove { encapsulatedEvent -= value; }
    }

    private int encapsulatedInteger;
    private event EventHandler<EventArgs> encapsulatedEvent;
}
```

要使用显式实现的接口，客户端代码引用的必须是一个接口的实例，而不能是接口实现类的实例。代码清单3-4展示了显式实现接口的引用方式。

代码清单3-4 显式实现接口时，实现接口的类实例是无法看到接口方法的

```
public class ExplicitInterfaceClient
{
    public ExplicitInterfaceClient(ExplicitInterfaceImplementation
        implementationReference, ISimpleInterface interfaceReference)
    {
        // Uncommenting this will cause compilation errors.
        //var instancePropertyValue =
        //implementationReference.ThisIntegerPropertyOnlyNeedsAGetter;
        //implementationReference.ThisMethodRequiresImplementation();
        //implementationReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        //implementationReference.InterfacesCanContainEventsToo += EventHandler;

        var interfacePropertyValue =
            interfaceReference.ThisIntegerPropertyOnlyNeedsAGetter;
        interfaceReference.ThisMethodRequiresImplementation();
        interfaceReference.ThisStringPropertyNeedsImplementingToo = "Hello";
        interfaceReference.InterfacesCanContainEventsToo += EventHandler;
    }

    void EventHandler(object sender, EventArgs e)
    {

    }
}
```

当实现接口的类与接口本身有方法签名冲突时，显式实现接口的方式是很有用的。

.NET Framework上下文中的所有方法都有自己的签名。方法签名是方法的标识，可以用来区别不同的方法，防止已有方法被覆写。方法签名包括方法名和参数列表。注意，.NET Framework方法的访问级别、返回值、抽象或者密封状态等都会影响最终的方法签名[①]。代码清单3-5展示了若干方法签名，其中一些会有冲突。如果两个方法的上述几个因素完全相同，那么它们的签名就会产生冲突。同一个类、接口或结构内的方法之间不允许存在签名冲突。

代码清单3-5　其中一些方法签名存在冲突

```
public class ClashingMethodSignatures
{
    public void MethodA()
    {

    }

    // This would cause a clash with the method above:
    //public void MethodA()
    //{
    //}

    // As would this: return values are not considered
    //public int MethodA()
    //{
    //    return 0;
    //}

    public int MethodB(int x)
    {
        return x;
    }

    // There is no clash here: the parameters differ.
    // This is an overload of the previous MethodB.
    public int MethodB(int x, int y)
    {
        return x + y;
    }
}
```

属性本身是没有参数列表的方法，所以只能用属性名称来区分。因此，如果两个属性名相同，属性签名就会产生冲突。

代码清单3-6中的类需要实现前面提及的接口`InterfaceOne`。

① 在方法重载的上下文中，方法签名不包括返回值。而在委托的上下文中，签名包括返回值，所以实现委托的方法要与委托的返回值类一致。——译者注

代码清单3-6　要实现接口的类会与接口本身产生方法签名冲突

```
public class ClassWithMethodSignatureClash
{
    public void MethodOne()
    {
    }
}
```

首先，因为方法签名相同，你只需要直接在类声明处加入实现接口的标记，如代码清单3-7所示。

代码清单3-7　隐式实现接口允许重用已经存在的方法

```
public class ClassWithMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
    }
}
```

客户端代码在任何时候调用这个类中的接口方法时，都会使用该类定义的同名方法。举个例子，要在Windows Forms中实现模型–视图–表示器（MVP）模式时，以及要添加需要`Close`方法才能在`Form`上实现的 `IView`接口时，这会很有用。代码清单3-8显示了其实际情形。

代码清单3-8　有时候可以巧妙地利用方法签名冲突

```
public interface IView
{
    void Close();
}
// . . .
public partial class Form1 : Form, IView
{
    public Form1()
    {
        InitializeComponent();
    }
}
```

然而，如果实现接口的类需要为接口方法提供不同的实现，就必须要显式实现接口的方法，从而避免出现两个签名冲突的方法体。代码清单3-9展示了这种情况。

代码清单3-9　显式实现接口方法以避免与类方法出现签名冲突

```
public class ClassAvoidingMethodSignatureClash : IInterfaceOne
{
    public void MethodOne()
    {
        // original implementation
```

```
    }

    void IInterfaceOne.MethodOne()
    {
        // new implementation
    }
}
```

类似地，如果一个类需要实现两个不相关但有同名方法的接口，你可以用同一个方法体来隐式实现这两个接口，或者显式实现两个接口，为两个接口分别提供一个独立的方法体。代码清单3-10展示了这种情况。

代码清单3-10　为两个接口实现各自的同名方法时，只有通过显式实现接口才能做到

```
public class ClassImplementingClashingInterfaces : IInterfaceOne, IAnotherInterfaceOne
{
    void IInterfaceOne.MethodOne()
    {

    }

    void IAnotherInterfaceOne.MethodOne()
    {

    }
}
```

3.1.3　多态

一个类型的对象可以隐式表现为另外一种不同类型的能力称为多态（polymorphism）。客户端代码看起来是与一种类型的对象交互，但该对象实际却是另外一种类型。这种高级手法是面向对象编程中解决问题的杀手锏之一，它是很多非常优雅且自适应强的解决方案的基础。

图3-3展示了一个抽象了交通工具行为的接口，三个分别为轿车、摩托车和快艇实现该接口的类。这三种交通工具有很大的差异，但是它们都实现了同一个接口定义的行为。

在这个示例里，假设交通工具有可以启停的引擎，可以转向，能够加速。多态能让引用`IVehicle`接口的客户端代码把所有具体类型都看作是相同的接口。至于轿车和摩托转向或加速的不同之处，或者快艇和火车引擎启动和停止的不同之处，都与调用交通工具接口的客户端代码无关。多态在编程开发中是一个极为有用的能力。在现实生活当中，当需要交通工具时，我们也都是交通工具接口的客户端。当然，上面示例中的接口定义比起实际的交通工具定义还差很远，不过原则都是一样的。只有知道汽车引擎工作原理的人才能开车吗？当然不是。是否清楚诸如引擎启停等工作原理的细节不会对驾驶人的开车技能有任何影响。这就是非常好的接口设计。

本章剩余部分会继续讲解有助于编写出自适应代码的设计模式和接口特性。多态是这些设计模式和接口特性的基础，它能让每个实现既定接口的类都变得有用，无论这些类已经写好或仍在

构思当中。

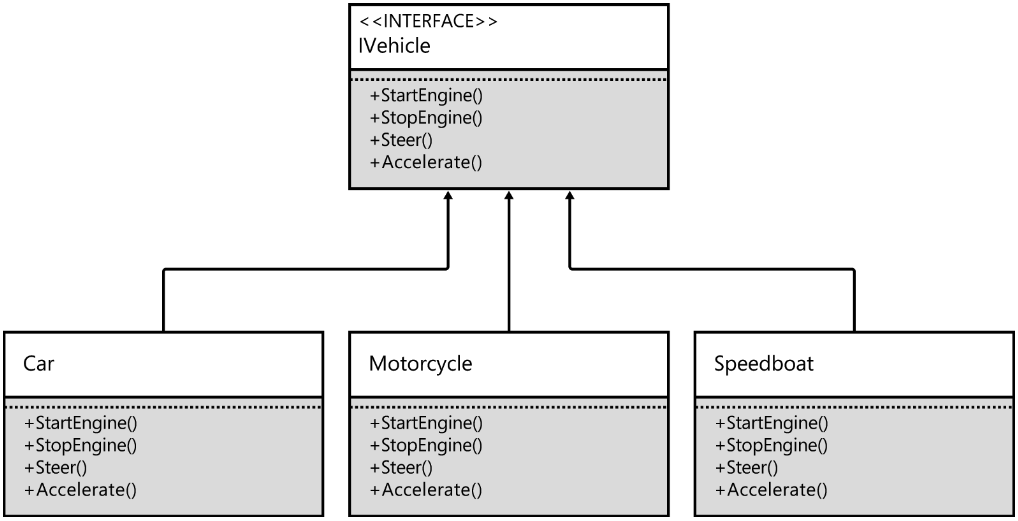


图3-3 接口把行为传递给所有实现它的类以形成多态的能力

3.2 自适应设计模式

"四人组"[①]所著的《设计模式：可复用面向对象软件的基础》一书让设计模式变得非常流行。虽然这本书已经出版了二十多年（对软件开发界而言，二十多年至少相当于四个冰河时代了），但在今天看来依然有着极高的价值。虽然其中有些模式已经被归类到反模式当中，但是其余的模式依然能够很大程度上提升代码的变更适应能力。

好的设计模式是接口和类之间可重用设计的精华，它们可以反复应用在很多不同的场景中，并且与项目、平台、语言和框架无关。与很多著名的最佳实践一样，了解设计模式这个理论工具要比对它一无所知要好得多。但是它们也可能会被滥用，也会不适合某些场合。有时，与简单方案相比，它会因为引入过多的类、接口、间接层和抽象而变得太过复杂。

经验告诉我，设计模式很容易被忽视或滥用。有些项目几乎没应用什么设计模式，代码也看不出明确的结构设计。另外一些项目则不受限制地使用设计模式，引入了过多的中间层和抽象，以至于得不偿失。正确的应用方式应该是在正确的场合使用正确的模式。

3.2.1 空对象模式

空对象模式是一种最常见的设计模式，也是一种最容易理解和实现的设计模式。它能让你避免引发意外的`NullReferenceException`异常，也无需到处检查对象是否为`null`。图3-4中的UML

① 四位作者分别是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides。

类图展示了空对象模式的应用方式。

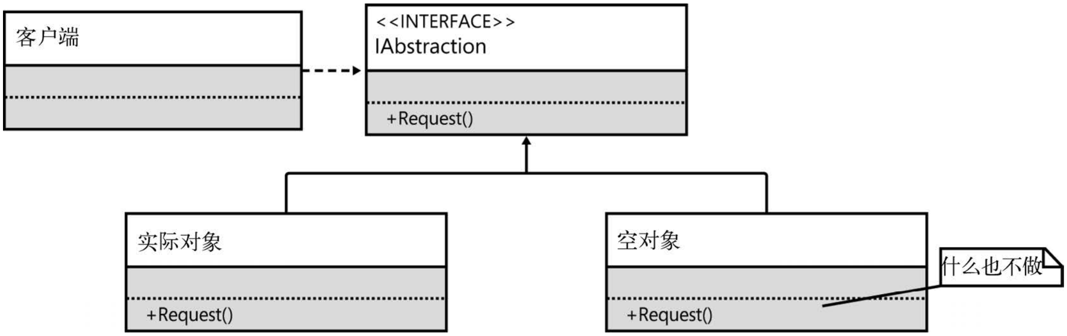


图3-4　使用UML类图表达空对象模式

代码清单3-11展示了典型的会引发NullReferenceException异常的代码。

代码清单3-11　如果不检查返回值是否为null，就有可能引发NullReferenceException异常

```
class Program
{
    static IUserRepository userRepository = new UserRepository();

    static void Main(string[] args)
    {
        var user = userRepository.GetByID(Guid.NewGuid());
        // Without the Null Object pattern, this line could throw an exception
        user.IncrementSessionTicket();
    }
}
```

所有调用IUserRepository.Get(Guid uniqueID)方法的客户端代码都有引发null引用异常的危险。在实践中，每个客户端都必须检查返回值是否为null，以避免解析null引用，因为后者会引发NullReferenceException异常。检查返回值是否为null的客户端代码如代码清单3-12所示。

代码清单3-12　检查值是否为null仍是所有客户端的职责吗

```
class Program
{
    static IUserRepository userRepository = new UserRepository();

    static void Main(string[] args)
    {
        var user = userRepository.GetByID(Guid.NewGuid());
        if(user != null)
        {
            user.IncrementSessionTicket();
        }
```

```
    }
}
```

检查对象值是否为空表明你让所有`IRespository`的客户端代码都做了太多不必要的工作。用这个方法的客户越多，忘记检查`null`引用的可能性越大。相反，你应该让产生问题的源头来完成检查工作，如下面的代码清单3-13所示。

代码清单3-13 服务代码应该实现空对象模式

```
public class UserRepository : IUserRepository
{
    public UserRepository()
    {
        users = new List<User>
        {
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid()),
            new User(Guid.NewGuid())
        };
    }

    public IUser GetByID(Guid userID)
    {
        IUser userFound = users.SingleOrDefault(user => user.ID == userID);
        if(userFound == null)
        {
            userFound = new NullUser();
        }
        return userFound;
    }

    private ICollection<User> users;
}
```

首先，这段代码试图从一个内存中的集合中获取指定ID的`User`，这与第2章中的实现是一样的。但是，现在你要做的是检查返回的`User`实例是否为`null`引用，如果是，你返回一个特殊的`IUser`的派生类型：`NullUser`。这个子类重写的`IncrementSessionTicket`方法实际上什么都没做，如代码清单3-14所示。实际上，正确的`NullUser`实现所重写的所有方法都应该尽量什么都不做。

代码清单3-14 `NullUser`方法实现什么都不做

```
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // do nothing
    }
}
```

此外，如果NullUser对象的属性或方法要返回另一个对象的引用，也应该返回这些类型的特殊空对象实现。换句话说，所有空对象实现都应该递归返回空对象实现。这样做后，客户端代码就无需检查null引用了。

这样做的另外一个好处就是，可以减少需要的单元测试数目。如果不应用空对象模式，每个客户端代码都要自己检查空引用，那么也就要有对应的单元测试来判断是否进行空对象测试。相反，会对存储库实现进行做单元测试以确保它返回NullUser实现。

IsNull属性反模式

有时候空对象模式会给接口引入一个名为IsNull的布尔型属性。该接口的所有具体实现都会使这个属性返回值不为真，只有接口的空对象实现返回值为真。代码清单3-15基于前面示例展示了IsNull属性。

代码清单3-15　只有接口的空对象实现的IsNull属性值为真

```
public interface IUser
{
    void IncrementSessionTicket();

    bool IsNull
    {
        get;
    }
}
// . . .
public class User : IUser
{
    // . . .
    public bool IsNull
    {
        get
        {
            return false;
        }
    }

    private DateTime sessionExpiry;
}
// . . .
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // do nothing
    }

    public bool IsNull
    {
        get
        {
            return true;
```

```
        }
    }
}
```

IsNull模式的问题在于它会让本应封装行为的对象暴露自身的逻辑。比如，需要在客户端代码中引入if语句来分别处理真正的实现和空对象实现。这违背了空对象模式的设计初衷：避免把自己的逻辑扩散到它的客户端代码中。代码清单3-16是这个问题的一个典型示例。

代码清单3-16　基于IsNull属性的逻辑使之成为一种反模式

```
static void Main(string[] args)
{
    var user = userRepository.GetByID(Guid.NewGuid());
    // Without the Null Object pattern, this line would throw an exception
    user.IncrementSessionTicket();

    string userName;
    if(!user.IsNull)
    {
        userName = user.Name;
    }
    else
    {
        userName = "unknown";
    }

    Console.WriteLine("The user's name is {0}", userName);

    Console.ReadKey();
}
```

通过让NullUser封装一个空用户名称，就可以去除上面示例中的不必要的if语句。如代码清单3-17所示。

代码清单3-17　经过合适的封装后，IsNull属性被废弃了

```
public class NullUser : IUser
{
    public void IncrementSessionTicket()
    {
        // do nothing
    }

    public string Name
    {
        get
        {
            return "unknown";
        }
    }
}
```

```
// . . .
static void Main(string[] args)
{
    var user = userRepository.GetByID(Guid.NewGuid());
    // Without the Null Object pattern, this line would throw an exception
    user.IncrementSessionTicket();

    Console.WriteLine("The user's name is {0}", user.Name);

    Console.ReadKey();
}
```


级联空值

C#语言有一个模拟Groovy[①]的特性，它提供了“级联空值”运算符。考虑下面的代码片段。

```
if(person != null && person.Address != null && person.Address.Country == "England")
{
    // . . .
}
```

它可以简化为以下代码。

```
if(person?.Address?.Country == "England")
{
    // . . .
}
```

因此，运算符?.成为了一种安全解析引用任何对象的方式，因为在最糟糕的情况下，也会有属性类型的default(T)来处理引用解析。我虽然不反对任何其他人或许认为有用的语法改进，但是拿?.与空对象实现作比较，我还是会选择后者，三个理由如下。

第一，很多情况下，只有一个简单的类默认值是不能满足需求的。上面示例中使用一个有意义的名称来避免引发NullReferenceException异常证明了Unknown不是一个默认值，而是对应用更有意义的数据。

第二，使用?.运算符还是会让所有客户端代码操心可能会出现null值。使用空对象模式的部分原因就是能够避免null检查并让客户端代码放心自由地解析引用。如果你选择级联空值语法，那么有可能让自己忘记去做引用解析。

第三，一个更主要的原因是，这种语法会在代码中泛滥成灾。偶尔使用int?来表示一个带有可选引用语法的int还可以接受，但是如果需要代码中每个引用解析的地方加上?.运算符，我想还是算了吧。

如果实现了适合的空对象实现，上面的代码可以像如下所示一样更简洁。

① Groovy是一种基于Java的动态编程语言（http://groovy.codehaus.org）。

```
if(person.Address.Country == "England")
{
    // . . .
}
```

这样编写代码的最大好处就是：客户端代码能专心完成自身的业务，根本不用担心可能会引发NullReferenceException异常。

3.2.2　适配器模式

适配器模式允许你为客户端提供一个接口的实例对象，而该对象并未真正实现这个接口。换句话说，适配器类实现了客户端期望的接口，但是接口方法的具体实现则委托给另外一个对象的不同方法来完成。这种模式通常用于目标类无法更改为期望接口的场合，因为目标类可能是密封的或者根本无法获得该类的源代码。适配器模式有两种实现方式：类适配器模式或对象适配器模式。

1. 类适配器模式

图3-5展示了类适配器模式中相互协作的类和接口

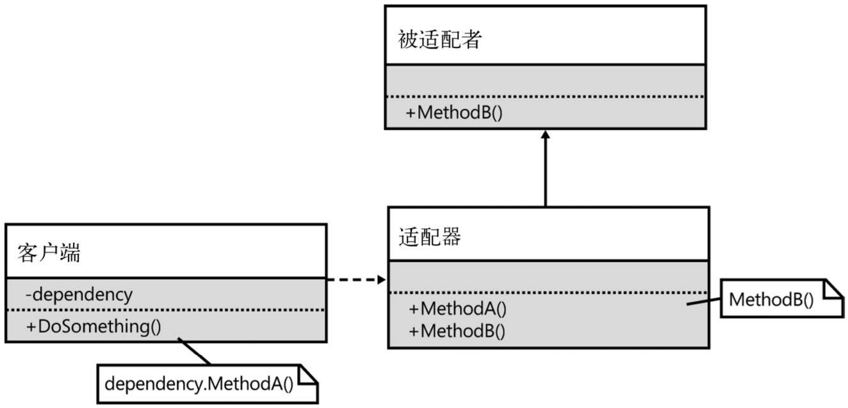


图3-5　类适配器的UML图

类设配器模式利用了适配器的继承，目标类的子类被适配为客户端所期望的接口。代码清单3-18展示了类适配器的实际应用。

代码清单3-18　类适配器模式利用了实现的继承关系

```
public class Adaptee
{
    public void MethodB()
    {

    }
}
```

```
// . . .
public class Adapter : Adaptee
{
    public void MethodA()
    {
        MethodB();
    }
}
// . . .
class Program
{
    static Adapter dependency = new Adapter();
    static void Main(string[] args)
    {
        dependency.MethodA();
    }
}
```

这种方式不如对象适配器模式常用，因为开发人员更倾向于使用组合而不是继承。这是因为继承是一种白盒（whitebox）重用，子类会依赖基类的实现而不是接口。组合则是一种黑盒（blackbox）重用，它把依赖局限在接口上，从而避免实现变更对客户端代码的影响。

2. 对象适配器模式

对象适配器模式使用组合委托一个黑盒实例对象来实现接口方法。图3-6展示了对象适配器模式中协作的类和接口。

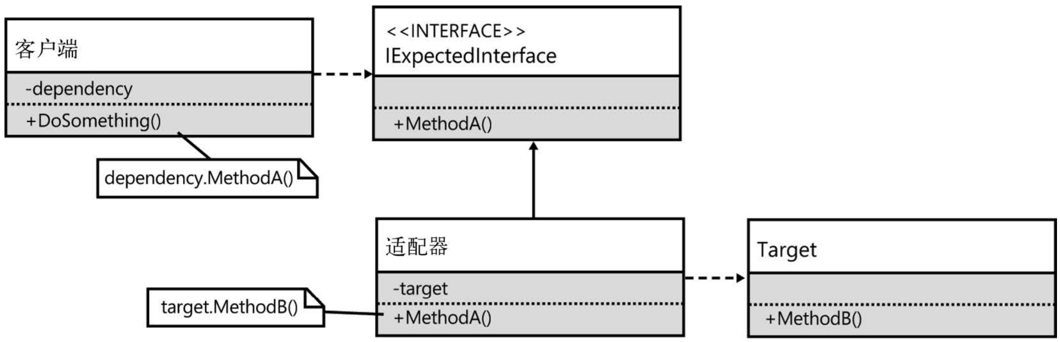


图3-6　对象适配器模式的UML图

代码清单3-19展示了对象适配器模式的应用示例。

代码清单3-19　适配器将目标类作为构造函数参数，并委托它实现接口

```
public interface IExpectedInterface
{
    void MethodA();
}
// . . .
public class Adapter : IExpectedInterface
{
```

```
        public Adapter(TargetClass target)
        {
            this.target = target;
        }

        public void MethodA()
        {
            target.MethodB();
        }

        private TargetClass target;
    }
    //
    public class TargetClass
    {
        public void MethodB()
        {

        }
    }
    // . . .
    class Program
    {
        static IExpectedInterface dependency = new Adapter(new TargetClass());
        static void Main(string[] args)
        {
            dependency.MethodA();
        }
    }
```

3.2.3　策略模式

策略模式能够在不需要重新编译的情况下（甚至在运行期间也可以）改变类的行为。图3-7中的UML图展示了策略模式的结构。

策略模式可以用于需要根据某个对象状态展示可变行为的类。如果类实例的行为会根据当前状态变化，此时使用策略模式封装这些可变行为就是最好不过的了。代码清单3-20展示了如何在一个类上创建和应用策略模式的代码。

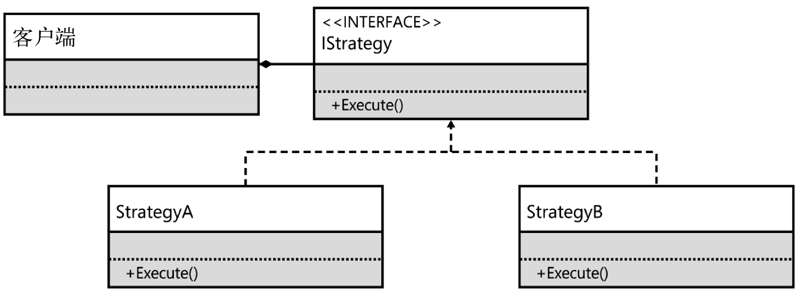


图3-7　策略模式的UML图

代码清单3-20 策略模式的用途

```
public interface IStrategy
{
    void Execute();
}
// . . .
public class ConcreteStrategyA : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyA.Execute()");
    }
}
// . . .
public class ConcreteStrategyB : IStrategy
{
    public void Execute()
    {
        Console.WriteLine("ConcreteStrategyB.Execute()");
    }
}
// . . .
public class Context
{
    public Context()
    {
        currentStrategy = strategyA;
    }

    public void DoSomething()
    {
        currentStrategy.Execute();

        // swap strategy with each call
        currentStrategy = (currentStrategy == strategyA) ? strategyB : strategyA;
    }

    private readonly IStrategy strategyA = new ConcreteStrategyA();
    private readonly IStrategy strategyB = new ConcreteStrategyB();

    private IStrategy currentStrategy;
}
```

`Context.DoSomething()`方法首先委托当前策略A做事，然后切换到另外一个策略B。下一次对该方法的调用会委托切换后的策略B做事，然后再次切换回原来的策略A，如此反复。

上面示例中，虽然选择策略的方式是包含在具体的方法实现中，但它并没有违背策略的实际作用：接口隐藏行为，接口的实现完成该行为定义的具体工作。

3.3 更多形式

不只是设计模式在利用接口的强大能力，还有其他更多特殊的接口应用方式也值得深入探讨。虽然这些方式并不是普适的，但是在一些特殊的场合它们会是最佳的选择。

与设计模式一样，滥用（overuse）这些特殊的接口应用方式会影响代码的可读性和可维护性。但是，实际应用中，很难为解决问题定出一个具体的使用模式和其他技术方法的最佳次数，因此在使用下面要介绍的这些方法时一定要谨慎。

3.3.1 鸭子类型

C#是一个静态类型语言，而鸭子类型（duck-typing）则是动态类型语言的一个特性，可以使用下面的鸭子测试（duck test）进行验证。

> 如果我看到一只鸟，它走起来像鸭子，游泳时像鸭子，叫起来也像鸭子，那么我就认为它是只鸭子。
>
> ——James Whitcomb Riley

在编程语言中，鸭子测试建议：只要对象展示了某个特性接口的行为，那么就应该看成该接口的实例。不幸的是，C#默认的情况并不是这样。如代码清单3-21所示。

代码清单3-21 尽管对象Swan实现了IDuck的所有方法，但实际上它不是一个IDuck

```
public interface IDuck
{
    void Walk();

    void Swim();

    void Quack();
}
// . . .
public class Swan
{
    public void Walk()
    {
        Console.WriteLine("The swan is walking.");
    }

    public void Swim()
    {
        Console.WriteLine("The swan can swim like a duck.");
    }

    public void Quack()
    {
        Console.WriteLine("The swan is quacking.");
    }
```

```
    }
    // . . .
    class Program
    {
        static void Main(string[] args)
        {
            var swan = new Swan();

            var swanAsDuck = swan as IDuck;

            if(swan is IDuck || swanAsDuck != null)
            {
                swanAsDuck.Walk();
                swanAsDuck.Swim();
                swanAsDuck.Quack();
            }
        }
    }
```


is谓词和as转换会分别返回false和null。公共语言运行时（CLR）并不会把Swan看作一个IDuck，即使Swan实际上实现了该接口。类型只能通过接口继承来实现接口。

有一些技巧能让Swan类在无需实现IDuck接口的情况下用作该接口的实例。你可以利用新版公共语言运行时中引入的动态类型特性，或使用一个名为Impromptu Interface的第三方库。

1. 使用动态语言运行时

从版本4开始，.NET Framework就不再是严格的静态类型平台了，它引入了dynamic关键字及其一些配套的支持类型以支持切换到动态类型的动态语言运行时（Dynamic Language Runtime，DLR）。下面的代码清单3-22展示了C#中的动态类型的一个例子。

代码清单3-22 动态语言运行时可以应用在鸭子类型上

```
    class Program
    {
        static void Main(string[] args)
        {
            var swan = new Swan();

            DoDuckLikeThings(swan);

            Console.ReadKey();
        }

        static void DoDuckLikeThings(dynamic duckish)
        {
            if (duckish != null)
            {
                duckish.Walk();
                duckish.Swim();
                duckish.Quack();
            }
        }
    }
```

然而，需要注意的是，示例中方法参数是dynamic类型的。为此，不仅你的代码需要以.NET Framework 4为目标框架，客户端代码也需要支持动态类型。让两者都支持动态类型，有时候不太现实。比如，你创建的所有方法全都采用dynamic参数，如果要这样的话，还不如直接选择诸如IronPython[1]等支持.NET Framework的动态类型语言。

2. 使用Impromptu Interface库

Impromptu Interface是一个.NET Framework库。使用NuGet安装好它后，就可以使用它提供的ActLike<T>()方法将传入的Swan类实例转化为一个IDuck接口实例。代码清单3-23展示了这种转换。

代码清单3-23　Impromptu Interface能够在C#中支持动态类型

```
class Program
{
    static void Main(string[] args)
    {
        var swan = new Swan();

        var swanAsDuck = Impromptu.ActLike<IDuck>(swan);

        if(swanAsDuck != null)
        {
            swanAsDuck.Walk();
            swanAsDuck.Swim();
            swanAsDuck.Quack();
        }

        Console.ReadKey();
    }
}
```

ActLike方法会在运行时会使用Reflection Emit创建一个新的类型。该类型会实现IDuck接口并将Swan类实例作为内部数据成员进行封装。任何对IDuck方法的调用，都会直接委托到该Swan类实例。Impromptu Interface实际上是在运行时应用对象适配器模式，这里要感谢强大的.NET Framework，因为它的反射特性支持能够在运行时创建新的类型。Impromptu Interface 是应用对象适配器模式的一种自动方式。

3. 公共语言运行时对鸭子类型的支持

有趣的是，公共语言运行时实际上有支持鸭子类型的能力，但只是应用在一个不大常用的场合：实现可枚举的对象。允许foreach循环枚举的类必须遵守一个特定的接口，但该接口的应用方式并不标准，目标类根本不需要显式继承于该接口，自己组织实现接口定义的方法即可。如代码清单3-24所示，Duck类允许foreach循环枚举。

① http://ironpython.net

代码清单3-24　CLR隐式对可枚举类支持鸭子类型

```
class Program
{
    static void Main(string[] args)
    {
        var duck = new Duck();
        foreach (var duckling in duck)
        {
            Console.WriteLine("Quack {0}", duckling);
        }

        Console.ReadKey();
    }
}
```

`Duck`类没有实现任何接口，但是`foreach`循环在编译时会去通过`Duck`实例调用`GetEnumerator()`方法，如图3-8所示。

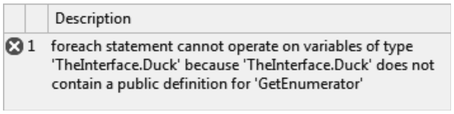


图3-8　类并没有被要求继承某个特定的接口，但是编译器会给出该类缺失一个公共方法的警告

在`Duck`类中实现了返回`void`的`GetEnumerator()`方法后，编译器还会继续给出缺失其他方法和属性的警告，如图3-9所示。

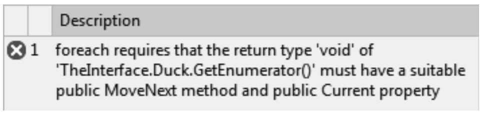


图3-9　`GetEnumerator`方法必须符合一个隐式的契约

代码清单3-25展示了这些要配合`GetEnumerator`方法实现的属性。

代码清单3-25　完成`DuckEnumerator`类的隐式接口

```
public class DuckEnumerator
{
    int i = 0;

    public bool MoveNext()
    {
        return i++ < 10;
    }

    public int Current
    {
```

```
            get
            {
                return i;
            }
        }
    }
```

至此，你已经成功地为`foreach`循环实现了所需的隐式接口。这就是已经存在的鸭子类型的应用，千真万确啊！

3.3.2　混合类型

混合类型（mixin）是一个从鸭子类型扩展出来的概念。该类包含了多个其他类的实现，但它并未继承这些类的实现。因为如前面所讲，C#并不支持继承多重实现，所以混合类型采用了其他技术方式。

可以使用扩展方法来实现混合类型，这种方案虽然小巧但也有局限性。扩展方法可以给已经定义好的类增加方法，在有些场景下很有用。另外一种方案是使用诸如Re-motion Re-Mix之类的第三方库，与Impromptu Interface类似，通过在运行时创建新类来支持你指定的所有接口，看起来就像是一个多功能适配器。

1. 使用扩展方法

从版本3.5开始，.NET Framework就提供了扩展方法的特性。它能够为已经存在的类添加新功能，既不需要访问原有类源代码，也不需要为该类标记`partial`。代码清单3-26展示了给已知接口增加两个新方法的示例。

代码清单3-26　扩展方法可以增强已有接口

```
public interface ITargetInterface
{
    void DoSomething();
}
// . . .
public static class MixinExtensions
{
    public static void FirstExtensionMethod(this ITargetInterface target)
    {
        Console.WriteLine("The first extension method was called.");
    }

    public static void SecondExtensionMethod(this ITargetInterface target)
    {
        Console.WriteLine("The second extension method was called.");
    }
}
```

在引用了`MixinExtensions`类后，客户端代码访问一个`ITargetInterface`实例时也会看到这两个扩展的方法。扩展方法的数目没有限制，也可以通过多个静态类为同一接口添加扩展方法。

代码清单3-27展示了`ITargetInterface`的另外两个带有参数的扩展方法。

代码清单3-27　扩展方法也可以带有参数

```
public static class MoreMixinExtensions
{
    public static void FurtherExtensionMethodA(this ITargetInterface target, int
        extraParameter)
    {
        Console.WriteLine("Further extension method A was called with argument {0}",
            extraParameter);
    }

    public static void FurtherExtensionMethodB(this ITargetInterface target, string
        stringParameter)
    {
        Console.WriteLine("Further extension method B was called with argument {0}",
            stringParameter);
    }
}
```

如代码清单3-28所示，这些扩展方法与接口的原有方法一样可以被任何客户端调用。

代码清单3-28　客户端可以访问接口的扩展方法

```
public class MixinClient
{
    public MixinClient(ITargetInterface target)
    {
        this.target = target;
    }

    public void Run()
    {
        target.DoSomething();
        target.FirstExtensionMethod();
        target.SecondExtensionMethod();
        target.FurtherExtensionMethodA(30);
        target.FurtherExtensionMethodB("Hello!");
    }

    private readonly ITargetInterface target;
}
```

使用扩展方法构造混合类型也有一些明显的局限。第一个是有关可测性的局限，如第4章会讲解到的，静态类很难被模拟和替换，所以会导致调用这些扩展方法的客户端代码较难测试。

更糟糕的是，静态类的扩展方法无法使用目标类实例的任何状态。当然，通过使用静态字典存储对象的状态值也可以变通取得实例状态，但是毕竟不完美，会带来其他问题。

另外一点值得注意的是，所有扩展方法都要指向相同的接口，所有运行实例都必需实现该接口。然而，真正的混合类型是一个聚合适配器，它应该能够同时实现多个不同接口。

2. 使用Re-motion Re-mix

实现混合类型的另外一种方式就是使用诸如Re-motion Re-mix之类的第三方库。在创建特定目标类时，Re-motion Re-mix能够通过运行时配置来指定要组合的类。与Impromptu Interface一样，Re-motion Re-mix会创建一个满足混合类型要求的所有接口的新类，而当调用接口方法时，这个新类的实例会将调用委托给混合类型的实例。代码清单3-29展示了多个接口及其实现。

代码清单3-29　多个独立的接口组合成一个混合类型

```
public interface ITargetInterface
{
    void DoSomething();
}
// . . .
public class TargetImplementation : ITargetInterface
{
    public void DoSomething()
    {
        Console.WriteLine("ITargetInterface.DoSomething()");
    }
}
// . . .
public interface IMixinInterfaceA
{
    void MethodA();
}
// . . .
public class MixinImplementationA : IMixinInterfaceA
{
    public void MethodA()
    {
        Console.WriteLine("IMixinInterfaceA.MethodA()");
    }
}
// . . .
public interface IMixinInterfaceB
{
    void MethodB(int parameter);
}
// . . .
public class MixinImplementationB : IMixinInterfaceB
{
    public void MethodB(int parameter)
    {
        Console.WriteLine("IMixinInterfaceB.MethodB({0})", parameter);
    }
}
// . . .
public interface IMixinInterfaceC
{
    void MethodC(string parameter);
}
// . . .
```

```
public class MixinImplementationC : IMixinInterfaceC
{
    public void MethodC(string parameter)
    {
        Console.WriteLine("IMixinInterfaceC.MethodC(\"{0}\")", parameter);
    }
}
```

注意，这里并没有一个单独的类同时实现了所有接口，而是通过配置Re-mix来请求一个TargetImplemation类的实例，该实例同时组合了所有接口和实现。代码清单3-30是一个配置示例。

代码示例3-30　使用Re-mix构造TargetImplementation类的实例

```
var config = MixinConfiguration.BuildFromActive()
    .ForClass<TargetImplementation>()
    .AddMixin<MixinImplementationA>()
    .AddMixin<MixinImplementationB>()
    .AddMixin<MixinImplementationC>()
    .BuildConfiguration();

MixinConfiguration.SetActiveConfiguration(config);
```

但是，你无法简单地通过new创建一个TargetImplementation来获得混合类型实例，而是必须通过Re-mix来完成。代码清单3-31展示了使用Re-mix创建实例的代码，幸运的是，代码还算小巧。

代码清单3-31　由Re-mix负责创建混合类型

```
ITargetInterface target = ObjectFactory.Create<TargetImplementation>(ParamList.Empty)
```

不幸的是，你不知道（也无法知道）ObjectFactroy.Create方法返回实例的确切类型，只知道它是子类（subclass）TargetImplementation的一个实例，这就是Re-mix的局限之一。这个局限会给使用混合类型的客户端代码带来负面影响，因为客户端代码在编译时唯一能确定的是TargetImplementation类实现了ITargetInterface接口，因此它不得不通过使用is来探测类，然后使用as将混合类型实例转换为想要的接口。代码清单3-32展示了这个问题。

代码清单3-32　类探测不值得提倡，但要使用混合类型就必须要使用它

```
public class MixinClient
{
    public MixinClient(ITargetInterface target)
    {
        this.target = target;
    }

    public void Run()
    {
        target.DoSomething();
```

```
        var targetAsMixinA = target as IMixinInterfaceA;
        if(targetAsMixinA != null)
        {
            targetAsMixinA.MethodA();
        }

        var targetAsMixinB = target as IMixinInterfaceB;
        if(targetAsMixinB != null)
        {
            targetAsMixinB.MethodB(30);
        }

        var targetAsMixinC = target as IMixinInterfaceC;
        if(targetAsMixinC != null)
        {
            targetAsMixinC.MethodC("Hello!");
        }
    }

    private readonly ITargetInterface target;
}
```

要使用混合类型，类探测最好是已经存在的或者必需的。很多库和平台也都在这样做。比如Prism（Windows Presentation Foundation/Model-View-ViewModel库）就使用了类探测，使用Prism的客户端类所需要的功能会被分割为多个独立的不同实现，然后再重新组合为一个混合类型。

3.3.3　流接口

流接口（fluent interface）会有一个或多个方法返回自身实例。这样，客户端代码就可以将所有返回实例的方法的调用链接起来，如代码清单3-33所示。

代码清单3-33　流接口支持方法链的调用方式

```
public class FluentClient
{
    public FluentClient(IFluentInterface fluent)
    {
        this.fluent = fluent;
    }

    public void Run()
    {
        // without using fluency
        fluent.DoSomething();
        fluent.DoSomethingElse();
        fluent.DoSomethingElse();
        fluent.DoSomething();
        fluent.ThisMethodIsNotFluent();

        // using fluency
```

```
        fluent.DoSomething()
            .DoSomethingElse()
            .DoSomethingElse()
            .DoSomething()
            .ThisMethodIsNotFluent();
    }

    private readonly IFluentInterface fluent;
}
```


这种方法链的调用方式可以改善代码的可读性，因为它可以避免重复引用接口实例。这种方式也变得越来越流行，用来实现配置或有限状态机。第8章中会有更多相关讲解。

实现流接口也很简单，只需要每个方法都返回类实例本身。因为类是接口的实现，所以返回的是接口实例，这样就保证隐藏了实现细节。代码清单3-34展示了IFluentInterface接口的定义以及实现。

代码清单3-34 实现一个简单的流接口很容易

```
public interface IFluentInterface
{
    IFluentInterface DoSomething();

    IFluentInterface DoSomethingElse();

    void ThisMethodIsNotFluent();
}
// . . .
public class FluentImplementation : IFluentInterface
{
    public IFluentInterface DoSomething()
    {
        return this;
    }

    public IFluentInterface DoSomethingElse()
    {
        return this;
    }

    public void ThisMethodIsNotFluent()
    {

    }
}
```

请注意，上面示例中有个接口方法不是流方法，因为它的返回类是void。返回非接口类数据的方法都不是流方法，非流方法会导致客户端代码中方法链方式的调用中断。

3.4 总结

本章告诉你接口是什么以及接口为什么对编写自适应代码如此重要。接口具有的多态能力支持在实现类中隐藏变化，这也是设计模式的基础。此外接口本身实际上什么都不做。

切记，没有相应的实现，接口本身也毫无用处。但是如果没有接口定义，实现类以及它们的各种依赖关系将会在代码中到处蔓延，从而严重影响代码的维护和扩展。良好组织和部署的接口就像一道防火墙，能把整洁有序的客户端代码和复杂烦乱的服务代码完整地隔离开来。

接口也有其他一些特殊特性，比如鸭子类型和混合类型。它们的使用频率不高，但是在合适的场合中，它们能够简化代码结构，同时能为代码的自适应能力提供一个额外的维度。

本章所讲解的内容为本书后续的各种接口应用奠定了坚实的基础。

第 4 章

单元测试和重构

完成本章学习之后，你将学到以下技能。

- 定义单元测试和重构，并且能够解释为什么二者都是很有用的软件开发技术。
- 理解单元测试与重构之间的内在联系。
- 以测试先行的方式编写代码，只有在测试需要时才集中实现。
- 重构产品代码来改善整体设计。
- 识别定义过度的单元测试并且重构它们。

本章会集中讲解单元测试和重构这两个不同但都是最佳编程实践的软件开发技术。

单元测试（unit testing）是指编写代码来专门测试其他代码。单元测试本身的源代码是可以编译和执行的。每个单元测试项在运行后，都会用简单的布尔值来报告成功与否，通常还会带有绿色或红色的图像指示。如果产品代码通过了所有单元测试，那么可以认为它基本上是可以工作的。只要有一个测试项运行失败了（即使是在成百上千的测试中），就可以肯定整个产品代码是无法正常工作的。

重构（refactoring）是一种逐渐改进现有代码设计的过程。这个过程类似于要编写很多次草稿代码，就像是我为了这本书也写了很多草稿一样。鉴于我们这些开发人员很少能一次就把事情做好的现状，重构能够让我们先从最简单的部分开始，然后逐步改进，最终实现一个比较好的方案。

单元测试能让随时随地重构成为可能。当你尽早开始单元测试时（也就是说，在编写任何产品代码前），就创建了一个安全网，它能捕获所有由后期重构引起的错误。如果一个本来成功的单元测试变成失败状态，你就知道最近对代码的改动带来了问题。这个边写代码边重构改进设计的方式是一个螺旋上升的过程，这个过程能够在逐步实现新特性的同时改善代码质量。

4.1　单元测试

一定程度上，单元测试应当是每个编程人员日常的必做功课之一。一些开发人员眼中的理想状态是整个产品代码（构成软件产品的基本代码）都是由测试驱动产生的，编写的所有测试都用来验证应用程序的行为。本章后面会讲解如何通过测试驱动开发来达到这个目的。但也要切记，我们的目标是要务实而不是成为纯粹主义者，在一个时间点上，接受一些可以后期解决的技术债

务并交付一些成果总比为了编写更多单元测试而迟迟无法交付任何成果要好得多。当然，每个项目对时效性和完成度的要求也有所不同。

应用一些已知的单元测试模式和原则能够保证有好的结果。这些模式和原则并不是新鲜事物，它们已经被普遍使用和验证过了。它们最关注的是如何布置和命名单元测试，特别是如何保证代码的可测试性。如果这些关注点被忽略了，源代码就会与单元测试脱节，失败的测试也变得无关紧要，由单元测试建立起来的安全网最终也会变得凋零破碎。

4.1.1　布置、动作和断言

每个单元测试都包括以下三个部分。

- 布置测试前置条件。
- 执行要测试的动作。
- 断言所期望的行为。

这三部分来源于布置、动作和断言（Arrange, Act, Assert，AAA）模式。你应该按照这个模式编写测试，这样其他人也能够理解它们。

> **注意**　有些读者可能更熟悉假设、当……时、应该会（Given, When, Then，GWT）模式。它与AAA很相似，只是描述测试的方式不同：假设某些前置条件已经得到满足，当执行测试的目标动作时，应该会发生预期的某些行为。

1. 布置测试前置条件

当要测试目标动作时，你必须先搭建测试场景的上下文。对于一些测试而言，你只需要简单地构造测试目标系统（System Under Test，SUT）的实例。这种情况下，测试目标系统就是你要测试的类型，如果类没有有效的实例，将无法测试类的任何方法。

代码清单4-1是个测试的布置代码示例，其中的`Account`类定义了客户的账户和交易。这里我使用的是Visual Studio内置支持的MSTest。后面还会在这个示例的基础上继续增加该测试的动作和断言部分。测试方法的名称是`AddingTransactionChangesBalance`，简明地描述了该测试的目的：确保用户账户上新发生的交易总是会改变账户账目以包括新发生的交易金额。

代码清单4-1　布置单元测试的第一步通常都是构造测试目标系统

```
[TestClass]
public class AccountTest
{
    [TestMethod]
    public void AddingTransactionChangesBalance()
    {
        // Arrange
        var account = new Account();
    }
}
```

这个示例中的布置过程很简单。整个测试的前置条件就是一个Account类的实例。你只需要在测试方法中直接调用new运算符创建一个实例。接下来你可以继续AAA模式的下一步了。

2. 执行可测试的动作

现在，能够执行动作的测试目标系统已经准备好了，你可以执行要测试的方法了。每个测试的动作阶段只应该与测试目标系统交互一次，比如只调用一个方法，或只使用一次属性的存或取。这样做的好处是，测试的执行路径足够简单清晰，编写和理解它也会很容易。

代码清单4-2展示了测试的动作部分。其中调用了account.AddTransaction方法，符合测试方法名称的含义。

代码清单4-2 每个动作执行部分应该与测试目标系统只有一次交互

```
[TestClass]
public class AccountTest
{
    [TestMethod]
    public void AddingTransactionChangesBalance()
    {
        // Arrange
        var account = new Account();

        // Act
        account.AddTransaction(200m);
    }
}
```


在该示例中，AddTransaction方法得到了一个传入参数值，它就是要给账户增加的交易资金额度。这里使用小数值意味着它有很高的精度，但是并没有指明货币单位。为了简单起见，我们暂时假设账户和交易都以美元为单位。

到此，这个测试已经包括了布置和动作部分，接下来你可以为它添加最后一个部分了。

3. 断言所期望的行为

紧跟上述两个步骤后是这个示例以及其他所有单元测试的关键点：断言。你能看到的绿色成功或红色失败图形标识就是断言的结果。所做的断言也与方法的名称含义相符，在这个示例中，名称的含义就是账户的账目被新的交易改变了。此处，断言要做的就是对实际值和期望值进行对比。根据测试目标系统的状态值进行断言是基于状态测试的一种常见形式，它会要求实际值和期望值完全相同。

从Account类的Balance属性中可以获得实际值，期望值则是由你定义的一个常量。这就意味着你必须提前知道期望值，这也是编写测试的关键点之一。你需要提前知道期望值，而不是通过代码推导出来。上面示例当中，得出这个期望值很容易。假设是一个新的账户，那么它的账目一定是0，如果你给账户增加200美元，那么期望值应该是多少呢？

$$\$0.00 + \$200.00 = \$200.00$$

因此，你可以编写断言部分来完成上面示例的AAA测试，如代码清单4-3所示。

代码清单4-3　添加了对期望行为的断言的单元测试

```
[TestClass]
public class AccountTest
{
    [TestMethod]
    public void AddingTransactionChangesBalance()
    {
        // Arrange
        var account = new Account();

        // Act
        account.AddTransaction(200m);

        // Assert
        Assert.AreEqual(200m, account.Balance);
    }
}
```

到这里，这个示例测试已经为运行作好了准备。通过运行测试，你可以验证测试目标系统是否像期望的那样起作用。

4. 运行测试

编写好单元测试后，你就可以使用一个单元测试运行器来运行它们。包含单元测试的测试工程的输出是无法直接执行的程序集，这就意味着测试工程无法执行本身而必须作为一个单元测试运行器的输入。Microsoft Visual Studio内置有测试运行器来支持符合MSTest定义的单元测试。如果是其他类型的单元测试，可能需要给Visual Studio安装测试运行器插件。

在Visual Studio当中，你可以使用Test>Run菜单项下的选项来运行MSTest单元测试。这里你先选择所有测试选项，它对应的快捷组合键是Ctrl+R+A。图4-1展示了运行符合AAA模式的测试后的输出。

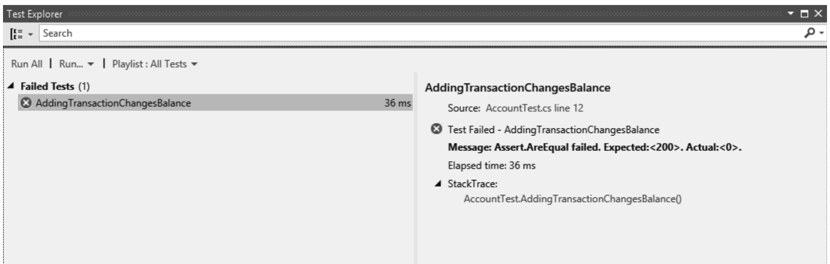


图4-1　使用Visual Studio内置的MSTest单元测试运行器运行单元测试

当你在左侧列表中选定一个单元测试时，右侧会显示出该测试的很多详细信息。示例中可以看到，该单元测试只耗费了短短的三十六毫秒，这恰恰就是编写单元测试的一大优势：即使运行成百上千的单元测试也不会耗费多少时间，相比之下，手动测试则会耗费大量的时间。

尽管如此，示例中的执行结果是失败的，因为期望值200并不等于实际值。断言语句中的

account.Balance属性的值是0。

这是因为Account类还缺少很多细节实现。代码清单4-4展示了到此所需的Account类的最小实现。

代码清单4-4 在单元测试前，测试目标系统并不需要具体实现

```
public class Account
{
    public void AddTransaction(decimal amount)
    {

    }

    public decimal Balance
    {
        get;
        private set;
    }
}
```

可以看得出来，AddTransaction方法什么都没有做，Balance属性也只是一个带有私有设置器的默认自动属性。为了让测试通过，你需要为Account类增加实现以达到你在断言中的期望。

4.1.2 测试驱动开发

为了实现某个单元测试，你无需完整实现整个测试目标系统。在测试驱动开发（Test-Driven Development，TDD）方法中，推荐在编写单元测试前不要有可以工作的测试目标系统。在使用测试驱动开发方法时，你需要先编写测试代码，然后才编写产品代码。产品代码中的每个类的每个方法都要经过一次失败的测试，这次失败的唯一原因就是产品的具体实现代码还不存在。断言要求的测试驱动开发以某种方式执行动作，但是这种动作还没有实现，因此测试失败了。在满足测试需求的前提下以尽可能简单的方式编写产品的实现代码后，就能看到该测试的绿色通过图标了。

1. 失败、成功、重构

到此，我们只介绍了AddingTransactionChangesBalance单元测试在失败、成功、重构（red, green, refactor）三步曲流程中的第一步。

(1) 针对测试目标系统的期望行为编写一个失败的测试。

(2) 给测试目标系统添加恰当的实现来让新加入的测试通过，且不影响所有已经存在的测试。

(3) 看看测试目标系统的设计或代码质量是否有改进的机会，如果有，立刻进行重构。

第一步创建的失败测试会在测试运行器上显示一个红色的失败图标。第二步加入实现后让测试从失败状态转变为成功状态，图标也会相应地变成绿色。到第三步时，就可以逐步递增地改善代码，同时无需担心变更会破坏现有功能。

要将测试状态从红变为绿（从失败转换为成功），需要按照流程定义的第二步动作：给测试

目标系统添加恰当的实现来让测试通过。因为上面示例中只有一个测试，所以你不需要担心影响到其他已有的成功测试。

测试所断言的行为是，在一个新账户上进行交易后，账户的余额应该从0变成了200。代码清单4-5展示了为了让测试通过所能作的最小改动。

代码清单4-5　总是引入最小实现来让失败的测试变得成功

```
public class Account
{
    public Account()
    {
        Balance = 200m;
    }

    public void AddTransaction(decimal amount)
    {

    }

    public decimal Balance
    {
        get;
        private set;
    }
}
```

示例中加粗显示了所作的最小代码改动。为了让测试由红变绿，直接在Account类的默认构造函数中将Balance属性初始化为200m。图4-2展示了Visual Sutido测试运行器重新运行失败测试后的截屏，它证明最小的改动起作用了，测试现在成功了。

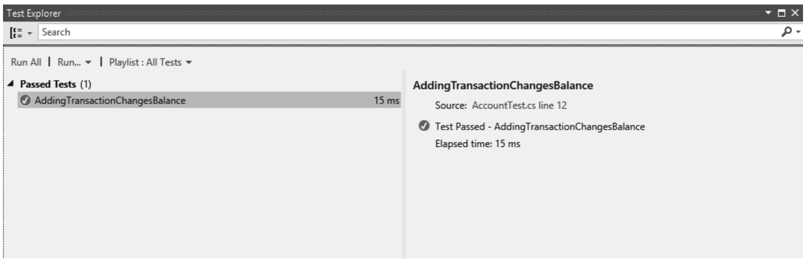


图4-2　现在，测试的确成功了，但是它真的正确吗

在你信心满满地准备开始重构前，应该清楚自己在第二步所做的实现是否正确。你可以通过增加另外一个单元测试来从另外一个期望角度来证明示例中的实现是不正确的。

要增加的测试定义了新创建账户对象上的期望余额。回顾上面的示例，在发生了一个200美元的交易后，Balance属性的期望值也是200美元，这是基于一个假设：新创建账号的余额是0美元。这也是一个期望的行为，所以你应该为此编写一个新的测试来断言实现代码是否正确。代码清单4-6展示了在新的单元测试上应用AAA模式后的代码。

代码清单4-6　下面示例中忽略了布置部分的代码

```
[TestMethod]
public void AccountsHaveAnOpeningBalanceOfZero()
{
    // Arrange

    // Act
    var account = new Account();

    // Assert
    Assert.AreEqual(0m, account.Balance);
}
```

首先，你可以看到这个新测试的名称也恰当地描述了断言所期望的行为。其次，你会看到这个单元测试示例中，布置部分代码是空的，这说明了AAA模式中的布置部分是可选的。这里要测试目标系统的行为就是默认构造函数，它就是该测试中动作部分与测试目标系统唯一交互的代码。最后的断言部分要求实现第一个示例的假设：新创建账户的账户余额应该是0美元。

尽管前一个单元测试依然是成功的，但是这个测试的运行结果是失败的，这就证明了为了让第一个测试通过而添加的最小实现是错误的。图4-3展示了MSTest运行器的输出。

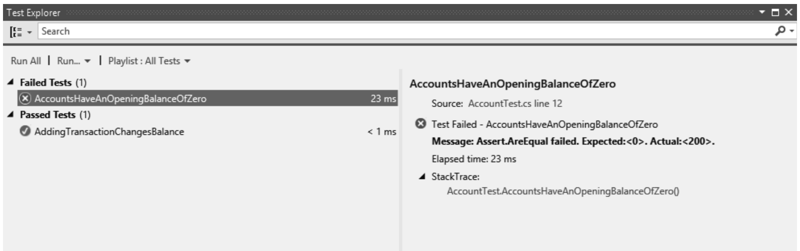


图4-3　为第一个单元测试添加的实现导致了第二个单元测试的失败

此时，如果你删除了默认构造函数中的实现代码，第二个验证开户账户余额（opening balance）的单元测试会由红变绿，但是同时第一个验证交易改变账户余额（adding transaction）的测试则会再次变回失败状态。这就证明了，更新后的实现对`Account`类的开户账户余额的测试是正确的，但对交易改变账户余额的测试来说是错误的。

增加新测试的另外一个限制就是需要改变测试目标系统的现有实现。每个新的测试都带有自己期望的行为，每个新的期望都要求在已有测试目标系统的内部实现上做出平衡。代码清单4-7展示了一种可能的最简单实现，它能够保证以上两个单元测试同时成功。

代码清单4-7　现在这个实现让两个测试都成功了，但是它真的是正确的吗

```
public class Account
{
    public void AddTransaction(decimal amount)
    {
```

```
            Balance = 200m;
        }

        public decimal Balance
        {
            get;
            private set;
        }
    }
```

至此，新开账户的账户余额是0，而且在调用AddTransaction方法后账户余额会变为200m。尽管两个测试都成功了，但是直觉告诉你这个实现一定是错误的。优先编写最简单的实现（而不是直接实现明显正确的方案）的关键点是根据你的直觉设计断言。你能编写出另外一个失败的单元测试来证明已有的实现是错误的吗？代码清单4-8展示了一个这样的示例。

代码清单4-8　这个单元测试与第一个单元测试的唯一区别是不一样的账户余额期望值

```
    [TestMethod]
    public void Adding100TransactionChangesBalance()
    {
        // Arrange
        var account = new Account();

        // Act
        account.AddTransaction(100m);

        // Assert
        Assert.AreEqual(100m, account.Balance);
    }
```

这个测试方法所做的工作与第一个单元测试一样，都是增加一个新的交易，只是交易值是100美元而不是200美元。尽管区别很小，但这个测试已经足以证明Account类的AddTransaction方法的现有实现是错误的。

按照单元测试三步曲的期望，这个测试第一次运行应该是失败的。如果这时把AddTransaction方法的实现硬编码为100m，这个测试会成功，但是第一个测试会再次变为失败的状态。此时，你可以实现一个对现有测试都正确的实现，如代码清单4-9所示。

代码清单4-9　下面的实现能保证三个单元测试都成功，但是它依然是错误的

```
    public class Account
    {
        public void AddTransaction(decimal amount)
        {
            Balance = amount;
        }

        public decimal Balance
        {
            get;
```

```
        private set;
    }
}
```

应用这个实现后三个单元测试都成功了，此时似乎已经大功告成了。但是还没有！现有的AddTransaction方法的实现依然不能满足实际的期望。代码清单4-10展示的第四个单元测试突出了现有实现的问题。

代码清单4-10　这个单元测试最终应该破解AddTransaction方法

```
[TestMethod]
public void AddingTwoTransactionsCreatesSummationBalance()
{
    // Arrange
    var account = new Account();
    account.AddTransaction(50m);

    // Act
    account.AddTransaction(75m);

    // Assert
    Assert.AreEqual(125m, account.Balance);
}
```


示例中的第四个单元测试最终让你发现AddTransaction方法的绝对正确的实现（至少对所有四个单元测试而言）。这种逐步重构方式的重点在于，随着需求的变更和新特性的引入，你需要整理所有类的所有期望以确保已经存在的所有单元测试都是成功的，而这些已经存在的单元测试就构成了一个安全网。没有这个安全网，虽然你能随意改动代码，小小的改动看起来影响范围不大而且你也做了手动验证，但是随后你会发现这些小小的改动很可能影响到了一些表面看起来毫无瓜葛的其他功能。

你添加的第四个测试断言账户上的余额应该是所有交易的总和。而现有的实现是直接将账户余额设置为最近交易的金额，这种不正确的实现一定会导致新的测试失败。代码清单4-11展示的AddTransaction方法的新实现能够保证所有四个单元测试都通过测试。

代码清单4-11　目前为止AddTransaction方法的最好实现

```
public class Account
{
    public void AddTransaction(decimal amount)
    {
        Balance += amount;
    }

    public decimal Balance
    {
        get;
        private set;
    }
}
```

当所有单元测试的状态都由红变绿时，你就可以开始重构以改善测试目标系统的现有实现了，只是当前这个示例非常简单，已经没有什么可以优化的了。对于要给测试目标系统新添加的测试而言，重构这一步非常关键。本章后续几节会更详细地讲解这种逐步重构的方式。

4.1.3　更复杂的测试

前面示例展示的单元测试的目标类型是一个应用程序域模型的一部分，该应用程序采用了测试先行的方式进行开发。正如第2章中所讲述的，这个域模型是处于用户界面层和数据访问层之间的业务逻辑层的一个具体实现。

1. 规格说明

接下来的一组测试依然是基于`Account`类的，但是测试目标是不同的业务逻辑层：服务。对于这个假想的应用程序而言，无论使用何种平台（ASP.NET MVC、WPF或Windows Form），这个服务都是可重用的。这就意味着该服务不依赖任何平台，而是基于`Account`类，尽管二者的关联是间接的。图4-4展示了构成该示例的层次和类之间的依赖关系。

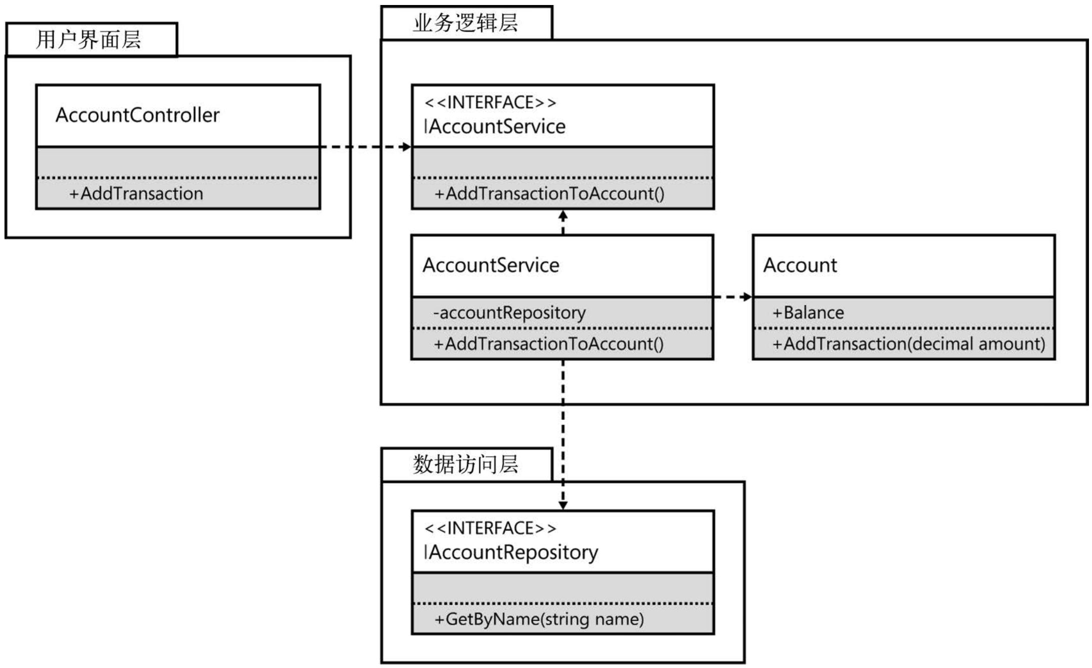


图4-4　依赖和实现构成了你将要测试的子系统

上面的UML图展示的三个包表达的是三层架构的方案。用户界面层包含了MVC控制器（Model-View-Controller， MVC），也可以使用MVVM（Model-View-ViewModel，MVVM）或者MVP（Model-View-Presenter，MVP）来替代MVC。具体来说，`AccountController`类会有处理用户界面在某个账户上进行交易的方法。这个控制器类依赖`IAccountService`接口，而

AccountService类则会实现该接口。

AccountService类和它的接口以及域模型Account类都属于业务逻辑层。注意，这里的包表达的是应用程序的逻辑层，而不是能直接映射到Microsoft Visual Studio当中的工程或程序集。这种分层方式采用的是阶梯模式，而不是随从反模式。AccountService类会需要以某种方式从你使用的持久存储中获取Account类的实例。因为你在业务逻辑层定义了域模型，所以数据访问层可以使用对象关系映射器（Object/Relational Mapper，ORM）来实现。

数据访问层的存储库接口用于隐藏具体的持久存储逻辑。IAccountRepository接口负责返回Account类的实例。服务依赖这个接口，因为该服务的实现需要检索Account类的实例。

2. 设计测试

与前面几节一样，采用测试驱动开发的方法以及布置、动作和断言模式来编写Account-Service类的AddTransactionToAccount方法的测试。首先，你需要想清楚对这个方法的期望：找到正确的Account类实例，委托它的AddTransaction方法，并把正确交易金额传入到该方法中。下面详细说明布置、动作和断言阶段。

- 布置：确保有可用的测试目标系统（AccountService类）的实例。
- 动作：调用它的AddTransactionToAccount方法。
- 断言：测试目标系统通过调用Account类实例的AddTransaction方法并把正确的交易金额传入。

代码清单4-12展示了该测试的第一个版本。

代码清单4-12 该测试的第一个版本

```
[TestClass]
public class AccountServiceTests
{
    [TestMethod]
    public void AddingTransactionToAccountDelegatesToAccountInstance()
    {
        // Arrange
        var sut = new AccountService();

        // Act
        sut.AddTransactionToAccount("Trading Account", 200m);

        // Assert
        Assert.Fail();
    }
}
```

断言前的代码看起来都没什么问题。断言就是调用某个对象的某个特定的方法并把某个具体的值传入，但是你又如何断言呢？此时就需要模拟了。

3. 使用临时实现测试

这个示例中，你首先要获得一个Account类的实例，然后才可以编写对该实例对象的断言。

因为AccountService类还需要使用IAccountRepository接口检索想要的Account类实例,所以你不能直接给AccountService类一个Account类的实例。相反，你需要给AccountService类一个IAccountRepository接口的实例，但是你并没有任何该接口的实现可用。此时就需要自己编写一个只用于测试该接口的临时实现类，因为你依赖的是接口而不是类型。代码清单4-13展示了这样的一个类，它本身属于单元测试程序集。

代码清单4-13　一个只用于测试的存储库接口的临时实现

```
public class FakeAccountRepository : IAccountRepository
{
    public FakeAccountRepository(Account account)
    {
        this.account = account;
    }

    public Account GetByName(string accountName)
    {
        return account;
    }

    private Account account;
}
```

现在，你可以编辑AccountService类实现从而能够提供这个临时存储库实例了。代码清单4-14展示了更新后的AccountService类的实现。

代码清单4-14　更新的AccountService类

```
public class AccountService : IAccountService
{
    public AccountService(IAccountRepository repository)
    {
        this.repository = repository;
    }

    public void AddTransactionToAccount(string uniqueAccountName, decimal transactionAmount)
    {

    }

    private readonly IAccountRepository repository;
}
```

现在能够完成这个单元测试了，只是更新了布置的标准。

- 确保有个可用的Account类实例来做断言。
- 确保为服务的构造函数传入一个可用的IAccountRepository接口的临时实现类的实例。

代码清单4-15中的失败测试包括了这些更新的标准和正确的断言代码。

代码清单4-15　这个测试肯定会失败，因为服务方法还没有任何具体实现

```
[TestClass]
public class AccountServiceTests
{
    [TestMethod]
    public void AddingTransactionToAccountDelegatesToAccountInstance()
    {
        // Arrange
        var account = new Account();
        var fakeRepository = new FakeAccountRepository(account);
        var sut = new AccountService(fakeRepository);

        // Act
        sut.AddTransactionToAccount("Trading Account", 200m);

        // Assert
        Assert.AreEqual(200m, account.Balance);
    }
}
```

首先你创建了一个账户余额为0的新账户。在此基础上，你创建了一个临时账户存储库实现类的实例。这个实例也实现了`IAccountRepository`接口，所以可以把它传入到`AccountService`类的构造函数中，这也是该测试中的测试目标系统。

在调用了要测试的目标方法后，你就可以断言该账户的余额是否变为了`200m`。图4-5展示了测试运行失败的截图，因为该方法还没有任何具体实现。

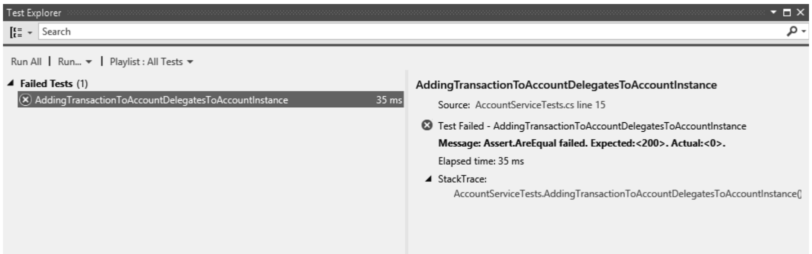


图4-5　在这个失败的测试基础上，需要继续对它进行改进以让红变绿

至此，你已经准备好了针对某些未实现行为的单元测试，可以从让该测试通过的最简单实现开始编写产品代码，如代码清单4-16所示。

代码清单4-16　`AccountService`类的一个能让测试通过的实现

```
public class AccountService : IAccountService
{
    public AccountService(IAccountRepository repository)
    {
        this.repository = repository;
    }
```

```
    public void AddTransactionToAccount(string uniqueAccountName, decimal
  transactionAmount)
    {
        var account = repository.GetByName(uniqueAccountName);
        account.AddTransaction(transactionAmount);
    }

    private readonly IAccountRepository repository;
}
```

单元测试驱动着你去做正确的实现。你必须使用存储库来获得账户，还必须调用该账户实例的AddTransaction方法来改变只读的Balance属性。如果后续有人对该单元测试的改动导致不能满足测试期望时，会立即看到失败的结果。

4. 使用模拟测试

如果再多想想，你就会认识到编写接口的临时实现类很快会变得不现实了。想想你要编写的单元测试的数目以及各种测试目标系统需要实现的各种不同的接口，你就明白仅仅为了支持你的单元测试，编写临时实现类会需要大量的编码工作。

还有另外一种模拟IAccountRepository接口的方式，但是这种方式需要外部模拟框架的支持。自己动手编写接口的临时实现类的好处在于无需引入第三方依赖，尽管如此，随着现在各种模拟框架的普及，引入一个这样的公开的第三方依赖是可以接受的。下面的示例使用了最流行的一个模拟框架：Moq，有时候也称为Moh-kyoo或Mok。

你可以使用NuGet快速检索Moq的在线包并将它添加到工程引用中。Moq的神奇之处在于它能动态创建任何你想模拟的接口的代理。代码清单4-17是用Moq模拟替代自己编写的临时实现类后的测试代码。

代码清单4-17　诸如Moq之类的模拟框架能轻松创建用于测试的替代实现

```
[TestMethod]
public void AddingTransactionToAccountDelegatesToAccountInstance()
{
    // Arrange
    var account = new Account();
    var mockRepository = new Mock<IAccountRepository>();
    mockRepository.Setup(r => r.GetByName("Trading Account")).Returns(account);
    var sut = new AccountService(mockRepository.Object);

    // Act
    sut.AddTransactionToAccount("Trading Account", 200m);

    // Assert
    Assert.AreEqual(200m, account.Balance);
}
```

上面示例中加粗的改动部分不再是初始化自己编写的库的临时实现类，而是创建了新的Mock<IAccountRepository>()对象。这个对象很强大，能够让你在要模拟的接口上设置各种期

望和行为。Mock类不会实现你的接口，因此，它不会像你的临时实现一样能直接得到一个IAccountRepository接口的实例。这是因为公共语言运行时并不允许从泛型参数继承。相反，被模拟者和所创建的代理实例之间是组合的关系。上面示例中传入到AccountService类构造函数中的Object属性提供了访问被模拟接口的方式。

你在给测试目标系统提供模拟对象前，需要定义该模拟对象的行为。Moq默认定义的是宽松模拟，这意味着模拟对象的所有返回都是default。任何引用类型的默认值都是null，这也适用于Account类。另外一种方式是严格模拟，对这种模拟对象上你没有预先定义的属性和方法的访问都会引起异常。无论选择哪种模拟方式，你都必须手动对创建的模拟对象设置期望的行为。

Mock实例的Setup方法设计得很巧妙。它能接受lambda表达式，表达式中的上下文参数就是被模拟类的实例。你可以通过调用被模拟类的方法并传入实际的参数来高效定义调用该方法时你所期望的行为。在不同的上下文中，期望的行为可以不同。Moq能让你在一个方法调用内设置以下期望值。

- 调用lambda表达式。
- 返回一个指定的值。
- 引发一个指定类型的异常。
- 确保该方法被调用。

对于当前这个示例，你所期望的是：返回一个指定的值。Mock.Setup方法的流接口允许在Return方法返回的Mock实例上进行链式调用。这种链式调用可以改善可读性并且避免测试的布置代码过于庞大。Return方法假设的前提是要指定返回的Account实例已经就绪。简单地说，你给Mock下达了以下指令。

> 如果在调用IAccountRepository实例的GetByName方法时传入值为Trading Account的账户名参数，请返回指定的Account类实例。

与前面的示例一样，再次运行这个测试的结果会是成功的。图4-6是运行结果截图。

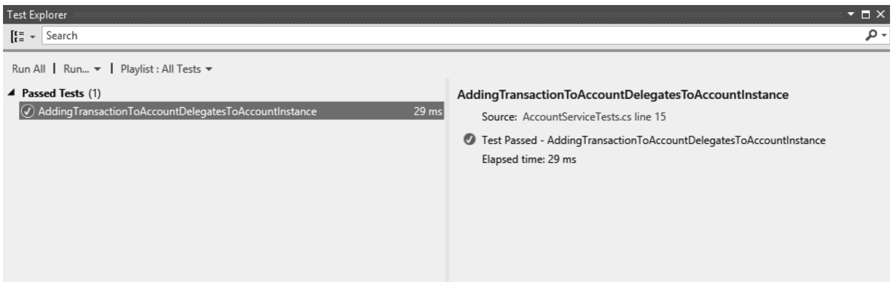


图4-6 在使用Moq模拟后，测试结果是成功的

在庆祝测试通过前，你需要承认你刚才欺骗了测试运行器。有时，你对某些失败的单元测试的编辑并不会让测试结果由红变绿。有时，无论你对某些成功的单元测试的编辑是什么，它始终是成功的状态。为了达到让单元测试先失败然后只有用正确的测试目标系统的实现才能让其通过

再次验证的目标，你应该先把AddTransactionToAccount方法的代码删除掉，然后恢复代码后再次验证以确认这些实现代码能让这个测试通过。这种编辑方式在单元测试流程中非常重要，它能规避出现假正确的现象，假正确是指实现上并不正确的单元测试也通过了验证。

模拟和过度测试

使用模拟测试这种实践很常见，但也会带来潜在的问题。因为基于模拟的测试很容易变得过度。过度规定的测试很脆弱，但你可以通过改变断言来避免出现这种现象。如果编写测试的人非常熟悉测试目标系统的内部原理，就容易出现这种现象，换句话说，过度规定的测试是针对测试目标系统的实现编写的，而不是针对测试目标系统所期望的行为编写的。

当然，使用模拟的单元测试需要知道测试目标系统是否已经实现，但是要切记单元测试规定的是所期望的行为，而不是实现细节，比如对测试目标系统可能要依赖的其他接口的调用等。如果你的断言是必须要调用某个接口的某个方法，那么这个测试已经关联了某个具体实现而不是某个特定行为。

过度规定的测试会阻碍对产品代码的重构。如果一组单元测试要与某个方法或者类同时出现，这就表明可以随意修改该方法或者类的实现。正确的测试只有在代码行为被破坏时才会失败，但是过度规定的测试无法提供这种保证，因为它们会在该方法和类的实现改变时失败，即使期望的行为依然是完整的。

有两种方式可以避免在使用模拟测试时出现过度规定的现象。第一种是只针对行为测试。基于状态的测试就是一种很好的只测试期望行为的方式。如果一个方法接受某个数据输入后返回该数据修改后的值，这个方法内部对测试而言就是黑盒。如果一个方法接受数据A、B和C后返回了数据X、Y和Z，那么测试不应当知道A、B和C是如何导出X、Y和Z的。在不影响单元测试的前提下，该方法后面会被修改。

第二种方式不常用，但是有时候不得不使用。你需要把单元测试和测试目标方法看作一个整体：两者必须同时改动。如果代码从不需要重构，你可以把单元测试和产品实现绑定在一起，也相当于承认了该单元测试是过度规定的。本书第二部分也会讲到，SOLID代码中会经常包含很多从不会修改的更直接的小规模类。

5. 更进一步的测试

第一步，你尝试了用测试驱动开发的方法成功完成一个可以工作的AccountService类。不过，该类定义中还有一些潜在的问题，这需要有更进一步的测试来确保类方法足够的健壮。到现在为止，上面的示例只要对代码的执行路径的测试没有发现问题和错误，你就觉得心满意足了。但是，实际上还是需要考虑以下这些其他因素。

- 如果账户存储库是个空引用？
- 如果在存储库中找不到指定名称的账户？
- 如果账户的方法引发了异常？

每当增加一个新的测试时，你要么在测试失败时无法确保覆盖实现的所有缺陷，要么在测试成功时相信自己的实现对正常代码执行路径和错误路径都是正确的。

在什么情况下账户存储库可能是空引用呢？这种情况只会发生在AccountService类的构造函数的输入参数也为空的情况下。因为有效的账户存储库需要依赖账户服务，所以你可以说这是该构造函数的前提（precondition）。为此，你可以像代码清单4-18这样编写测试。

代码清单4-18 没有布置和断言的测试是一种测试异常的有效模式

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void CannotCreateAccountServiceWithNullAccountRepository()
{
    // Arrange

    // Act
    new AccountService(null);

    // Assert
}
```

这个测试与前面的测试的断言方式不一样。MSTest为这种测试提供了ExpectedException-Attribute属性，该属性的参数代表了你期望的异常类型。上面示例中的代表了你期望AccountService类的构造函数的输入参数是一个IAccountRepository实例的null引用时应该引发一个ArgumentNullException类的异常。这个新的测试就是你需要的前置条件，它能保证其他使用账户服务的测试方法总是有个有效的实例，因此也无需再去做null引用的判断和处理。这个方法会像期望的那样失败，具体信息如下所示。

```
Test method
ServiceTests.AccountServiceTests.CannotCreateAccountServiceWithNullAccountRepository
  did not throw an exception. An exception was expected by attribute
  Microsoft.VisualStudio.TestTools.UnitTesting.ExpectedExceptionAttribute defined on the test
  method.
```

为了让这个测试通过，你需要在产品代码中实现前置条件检测。代码清单4-19给出了一种实现方式。

代码清单4-19 给构造函数传入空的账户存储库会引发异常

```
public AccountService(IAccountRepository repository)
{
    if (repository == null) throw new ArgumentNullException("repository", "A valid account
  repository must be supplied.");

    this.repository = repository;
}
```

示例中加粗的就是新增的代码。这种方式可以确保你尽快失败。如果不设置这种前置条件，

最终会引发的是一个NullReferenceException异常，它会发生在你第一次尝试访问这个null存储库实例的时候。

在构造函数测试通过后，你就可以处理在存储库中找不到匹配账户的情况了。假设你的存储库代码并没有实现空对象模式，这种模式在第3章中介绍过，它能够避免返回null对象或者在存储库找不到请求的对象时引发异常。相反，这个示例中，你的存储库代码应该返回一个账户的null引用。代码清单4-20展示了一个强制出现这种期望行为的单元测试。

代码清单4-20　既没有期望异常属性也没有断言的单元测试

```
[TestMethod]
public void DoNotThrowWhenAccountIsNotFound()
{
    // Arrange
    var mockRepository = new Mock<IAccountRepository>();
    var sut = new AccountService(mockRepository.Object);

    // Act
    sut.AddTransactionToAccount("Trading Account", 100m);

    // Assert
}
```

这个新的单元测试中断言部分是空白的，也没有ExceptedException属性。这是因为你的期望就是动作部分代码不应该引发任何异常。如果有异常引发，那么测试就失败了，如果测试代码没有任何异常引发（也没有其他可能导致失败的断言），那么该测试默认就是成功的。

示例测试的布置阶段，模拟了存储库并把它传给测试目标系统（提供了一个有效的存储库实例来满足前置条件）但是没有对它设置期望的行为。这就意味着对IAccountRepository.GetByName()方法的调用会返回null。然后在返回对象的基础上尝试调用Account.AddTransaction()方法。因为是个null实例，所以会引起一个NullReferenceException异常并导致测试失败。为了让该测试由红变绿，你需要在方法实现中引发该异常，如代码清单4-21所示。

代码清单4-21　使用if语句防止方法引发NullReferenceException异常

```
public void AddTransactionToAccount(string uniqueAccountName, decimal transactionAmount)
{
    var account = repository.GetByName(uniqueAccountName);
    if (account != null)
    {
        account.AddTransaction(transactionAmount);
    }
}
```

通过增加一条简单的if语句来在使用前先判断账户对象是否为null，就可以防止该方法引发NullReferenceException异常了，同时也保证了对应的单元测试是成功的。

最后一个需要增加的单元测试是针对当账户的AddTransaction方法引发异常时账户服务的期望行为而设计的。为了避免分层间的依赖泄漏，一个好的实践方式是在当前层用一个新的异常来封装从底层引发的异常。图4-7展示了这个方式。

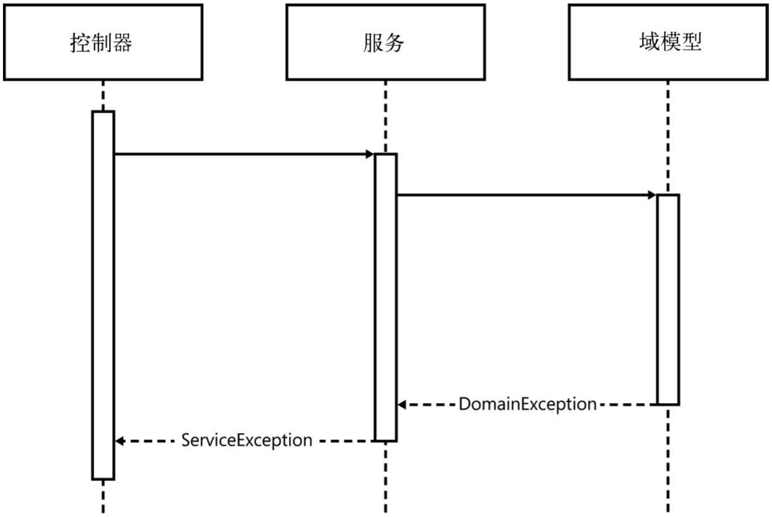


图4-7 每个层都定义自己的异常类型来封装来自更低层的异常

域模型引发的异常是针对域模型设计的，如果服务层允许这种低层次的异常引发给控制器，这会要求控制器清楚DomainException类以便高效地捕获并处理这类异常。这明显会引入你并不想看到的控制器和域模型层间的依赖关系。相反，服务层应该自己负责捕获域模型的异常并将其封装在ServiceException异常实例中，然后才引发给控制器。因为控制器本身是依赖服务层次的，因此它可以，也应该，处理服务层引发的任何异常。这里要清楚的一点是，DomainException类要作为ServiceException类的内部成员，如果不这样做，你就会失去很有价值的引发原始异常的上下文信息。代码清单4-22展示了确保你的多个类协同实现这种期望行为的单元测试。

代码清单4-22 调用模拟的账户时会引发异常

```
[TestMethod]
[ExpectedException(typeof(ServiceException))]
public void AccountExceptionsAreWrappedInThrowServiceException()
{
    // Arrange
    var account = new Mock<Account>();
    account.Setup(a => a.AddTransaction(100m)).Throws<DomainException>();
    var mockRepository = new Mock<IAccountRepository>();
    mockRepository.Setup(r => r.GetByName("Trading Account")).Returns(account.Object);
    var sut = new AccountService(mockRepository.Object);

    // Act
    sut.AddTransactionToAccount("Trading Account", 100m);
```

```
        // Assert
    }
```

示例中的期望异常属性用于断言测试目标系统是否引发了一个ServiceException异常，而代码中模拟的账户实例则会去引发一个DomainException异常。因此，由测试目标系统负责异常之间的转换。现在你的产品代码中的方法还没有进行这个异常转换，所以测试会像期望的那样失败，具体信息如下所示。

```
Test method
ServiceTests.AccountServiceTests.AccountExceptionsAreWrappedInThrowServiceException
  threw exception Domain.DomainException, but exception Services.ServiceException was expected.
  Exception message: Domain.DomainException: Exception of type 'Domain.DomainException' was
  thrown.
```

期望异常属性会判断引发的异常是否是指定的异常类型。代码清单4-23展示了需要在AddTransactionToAccount方法中所做的改动。

代码清单4-23　引入的try/catch代码块中包括了异常的转换

```
public void AddTransactionToAccount(string uniqueAccountName, decimal transactionAmount)
{
    var account = repository.GetByName(uniqueAccountName);
    if (account != null)
    {
        try
        {
            account.AddTransaction(transactionAmount);
        }
        catch(DomainException)
        {
            throw new ServiceException();
        }
    }
}
```

尽管引入try/catch代码块能让上面的测试由红变绿，但是ServiceException异常并没有包含DomainException异常，也就是说，测试工作依然没有完成。

6. 为修复缺陷编写测试

想象一下你收到了一个现有示例代码的缺陷报告。该报告中有下面这样一句描述。

> 我在我的账号上做交易时收到了一个ServiceException异常。

于是你着手去重现这个问题并发现了引发的异常，这也是问题最可能的根源。但是，ServiceException异常取代了DomainException异常，因此很难理解问题的最根本原因。前面的代码中，原有的异常并没有被包含在新的异常中，因为你的单元测试中并没有断言来确保这一点。

当你收到这样的缺陷报告时，应该做的第一件事就是编写一个要失败的单元测试来捕获两件事：确保缺陷发生的必需的具体复现步骤以及旧的单元测试遗漏的期望行为。代码清单4-24中的

单元测试包含了需要完成的两件事。

代码清单4-24 手动断言要引发的异常的示例

```
[TestMethod]
public void AccountExceptionsAreWrappedInThrowServiceException()
{
    // Arrange
    var account = new Mock<Account>();
    account.Setup(a => a.AddTransaction(100m)).Throws<DomainException>();
    var mockRepository = new Mock<IAccountRepository>();
    mockRepository.Setup(r => r.GetByName("Trading Account")).Returns(account.Object);
    var sut = new AccountService(mockRepository.Object);

    // Act
    try
    {
        sut.AddTransactionToAccount("Trading Account", 100m);
    }
    catch(ServiceException serviceException)
    {
        // Assert
        Assert.IsInstanceOfType(serviceException.InnerException, typeof(DomainException));
    }
}
```


对于这个测试，单单使用ExpectedException属性是不够的。你需要自己编写代码来断言引发ServiceException异常的InnerException成员是否为DomainException。这个断言可以证明你已经封装了域异常并保留了原始错误发生的上下文信息。所有软件缺陷都可以看作对应单元测试的缺失：期望行为的规范不够完整。代码清单4-25展示了让该单元测试通过验证对产品代码所做的修改。

代码清单4-25 新的异常封装了原始异常

```
public void AddTransactionToAccount(string uniqueAccountName, decimal transactionAmount)
{
    var account = repository.GetByName(uniqueAccountName);
    if (account != null)
    {
        try
        {
            account.AddTransaction(transactionAmount);
        }
        catch(DomainException domainException)
        {
            throw new ServiceException("An exception was thrown by a domain object",
  domainException);
        }
    }
}
```

通过这个测试后，你可以再次尝试重现报告中提及的异常，这一次，你可以判断问题的根本原因了。

7. 测试初始化

我们一起回顾一下目前你所写的所有测试。每一个都逐步变得越来越复杂，需要加入更多代码来设定你的期望。如果你能整理出公共测试代码并缩短这些测试代码，这样也很好。与其他单元测试框架一样，MSTest允许你在测试类型内为所有测试方法编写公共的初始化方法。你可以为这个初始化方法任意命名，但是必须要带有TestInitialize属性。

初始化方法包括了几乎所有单元测试都需要的公共代码：初始化模拟对象。你可以把模拟对象保存在测试类型的私有数据对象中，这样每个测试方法都可以访问它们。同样，也可以将测试系统存放在测试类型的私有数据中，因为上面的示例中，测试目标系统除了在构造函数需要模拟的存储库实例作为输入参数外，它并不依赖其他任何因单元测试而异的元素。代码清单4-26展示了在测试类型中支持初始化方法所需要的改动。

代码清单4-26　可以在初始化方法中构造模拟对象和测试目标系统

```
[TestClass]
public class AccountServiceTests
{

    [TestInitialize]
    public void Setup()
    {
        mockAccount = new Mock<Account>();
        mockRepository = new Mock<IAccountRepository>();
        sut = new AccountService(mockRepository.Object);
    }

    private Mock<Account> mockAccount;
    private Mock<IAccountRepository> mockRepository;
    private AccountService sut;
}
```

每个测试方法都会单独调用构造这些模拟对象和测试目标系统的初始化方法，这样你就可以通过移除对象构造代码来简化某些单元测试。代码清单4-27展示了对最后编写的Account-ExceptionAreWrappedInThrowServiceException测试方法的改动。

代码清单4-27　简化后的测试方法代码更短且更易阅读

```
[TestMethod]
public void AccountExceptionsAreWrappedInThrowServiceException()
{
    // Arrange
    mockAccount.Setup(a => a.AddTransaction(100m)).Throws<DomainException>();
    mockRepository.Setup(r => r.GetByName("Trading Account")).Returns(mockAccount.Object);

    // Act
```

```
        try
        {
            sut.AddTransactionToAccount("Trading Account", 100m);
        }
        catch(ServiceException serviceException)
        {
            // Assert
            Assert.IsInstanceOfType(serviceException.InnerException, typeof(DomainException));
        }
    }
```

虽然在一个方法内只移除了三行代码，看起来作用并没有那么明显，但是让所有的单元测试都变得简短的效果累积后就能得到更高的可读性。众所周知，按照约定，示例代码中变量名的mock前缀代表了该变量是模拟对象的实例，而sut变量则代表了你要测试目标系统的实例。

4.2 重构

如果你按照测试驱动开发流程在实现期望行为前先编写一个注定失败的单元测试，你的代码就会更加健壮。尽管如此，代码的组织也许会变得不易理解。在编码过程中，很多时候都应该停下手头的单元测试和产品代码开发，从而集中精力去重构已有的代码。

重构（refactoring）是一个改善现有代码设计的过程。每次重构的规模和范围会有不同，花半分钟给变量改一个更明确的名称是一次重构，必要时将用户界面层逻辑和域逻辑分离开，这样影响巨大的架构改动也是一次重构。

4.2.1 更改已有代码

接下来，你会通过实践逐步改善同一个类型的代码，每一步的改动都有着既定的目标。Account类依然是重构的目标类型，不过这次它有了一个新的CalculateRewardPoints方法。与很多公司一样，你的客户也希望通过积分方式给忠实的用户回馈一些奖励。在满足一些标准后，客户可以获得一定的奖励积分。代码清单4-28展示了新的Account类。

代码清单4-28 新的Account类在账户余额的基础上增加了奖励积分功能

```
public class Account
{
    public Account(AccountType type)
    {
        this.type = type;
    }

    public decimal Balance
    {
        get;
        private set;
    }
```

```
    public int RewardPoints
    {
        get;
        private set;
    }

    public void AddTransaction(decimal amount)
    {
        RewardPoints += CalculateRewardPoints(amount);
        Balance += amount;
    }


    public int CalculateRewardPoints(decimal amount)
    {
        int points;
        switch(type)
        {
            case AccountType.Silver:
                points = (int)decimal.Floor(amount / 10);
                break;
            case AccountType.Gold:
                points = (int)decimal.Floor((Balance / 10000 * 5) + (amount / 5));
                break;
            case AccountType.Platinum:
                points = (int)decimal.Ceiling((Balance / 10000 * 10) + (amount / 2));
                break;
            default:
                points = 0;
                break;
        }
        return Math.Max(points, 0);
    }

    private readonly AccountType type;
}
```

下面列举一下该类的重要修改。

- 增加了一个新的属性，用来追踪客户账户下相关的奖励积分。
- 每个账户都有一个私有的类型数据来表明该账户的级别：银卡、金卡或者白金卡。
- 当账户上发生交易时，客户可以获得一定的奖励积分。
- 获得的奖励积分的额度取决于计算方法中使用的以下几个因素。
  - 账户类型——账户级别越高，获得的积分越多。
  - 交易额度——客户消费越多，获得的积分越多。
  - 账户余额——金卡和白金卡能获得更多积分，以鼓励客户保持较高的账户余额。

假设这些代码和相应的单元测试都一起写好了，这些测试能更好地保证对代码的修改不影响它们声明的行为。这一点非常重要：重构只改变代码的布局而不是输出。如果你尝试在没有单元测试的情况下重构现有代码，你又如何保证你的重构没有破坏期望的行为呢？你将无法快速得到

失败的提示，而只能在后期运行代码时发现问题，比如测试时或者是更糟糕的情况——部署后。

1. 用常量替代魔法数

第一个重构很简单但是对改善代码可读性有着至关重要的作用。示例中的Calculate-RewardPoints方法中包含了很多魔法数。方法中使用了六个不同的数字，而且没有它们的任何上下文信息，比如它们是什么以及为什么需要它们等。编写这些代码的人知道这些数字的意义，但是这也只可能维持一到两周，更长时间后，他们也会忘记数字5或数字2的原始意义了。代码清单4-29展示了重构后的代码。

代码清单4-29　重构后的代码对不熟悉代码的人来说可读性更高

```
public class Account
{
    public int CalculateRewardPoints(decimal amount)
    {
        int points;
        switch(type)
        {
            case AccountType.Silver:
                points = (int)decimal.Floor(amount / SilverTransactionCostPerPoint);
                break;
            case AccountType.Gold:
                points = (int)decimal.Floor((Balance / GoldBalanceCostPerPoint) + (amount/
    GoldTransactionCostPerPoint));
                break;
            case AccountType.Platinum:
                points = (int)decimal.Ceiling((Balance / PlatinumBalanceCostPerPoint) +
  (amount / PlatinumTransactionCostPerPoint));
                break;
            default:
                points = 0;
                break;
        }
        return Math.Max(points, 0);
    }

    private const int SilverTransactionCostPerPoint = 10;
    private const int GoldTransactionCostPerPoint = 5;
    private const int PlatinumTransactionCostPerPoint = 2;

    private const int GoldBalanceCostPerPoint = 2000;
    private const int PlatinumBalanceCostPerPoint = 1000;
}
```

所有魔法数都被相应的常量替代。其中三个常量是与三个账户类型下每个积分需要的交易额度有关，另外两个是与金卡以及白金卡下每个积分需要的账户余额数值有关。

这个重构的好处就是不熟悉代码的人也能很容易理解代码的含义了，因为这些魔法数有了相应的自描述变量名。只用没有意义的变量名A、B或X来替代魔法数起不到任何改进可读性的作用。因此要尽量选择能够精确表达意图的变量名，而且不要怕名称太长。要抓住每个给变量、类和方

法命名的机会来让代码变得具有自描述的能力。

2. 用多形性替代条件表达式

接下来的这个重构更加复杂一些。CalculateRewards方法内的switch语句会根据账户类型切换不同的计算积分的算法，从两方面来看这都会产生问题。第一，它会影响代码的可读性，更重要的是，它还会引入更多的维护负担。不难想象，不久的将来很有可能出现一个新的账户类型。假设很多人都满足不了银卡的要求，所以你需要引入一个新的类型：青铜卡。为了增加这个新的青铜卡类型，你需要更改现有的Account类代码并进行相关测试。如果代码验证完成并且已经部署，应该尽量避免修改它们。相反，应该尝试其他方式来扩展代码，以达到不改变现有代码的前提下满足新的需求变更。

这里的目标就是能更容易地增加一个新的账户类型且能改善代码的可读性。为此，你需要借助多形性的强大能力。你需要对各种类型的账户建模并继承自Account类。金卡账户类型就有了自己的GoldAccount类定义，同样，也有银卡的SilverAccount类和白金卡的PlatinumAccount类。

第一步先定义这些具体的账户类型，如代码清单4-30所示。

代码清单4-30　每个账户类型现在都有自己的类定义

```
public class SilverAccount
{
    public int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Floor(amount / SilverTransactionCostPerPoint), 0);
    }

    private const int SilverTransactionCostPerPoint = 10;
}
// . . .
public class GoldAccount
{

    public decimal Balance
    {
        get;
        set;
    }

    public int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Floor((Balance / GoldBalanceCostPerPoint) + (amount /
  GoldTransactionCostPerPoint)), 0);
    }

    private const int GoldTransactionCostPerPoint = 5;
    private const int GoldBalanceCostPerPoint = 2000;
}
// . . .
public class PlatinumAccount
```

```
{
    public decimal Balance
    {
        get;
        set;
    }

    public int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Ceiling((Balance / PlatinumBalanceCostPerPoint) +
(amount / PlatinumTransactionCostPerPoint)), 0);
    }

    private const int PlatinumTransactionCostPerPoint = 2;
    private const int PlatinumBalanceCostPerPoint = 1000;
}
```


注意，到这一步，原有的Account类还没有变，这些具体的账户类型都是独立的。相应地，这些类型的单元测试都会复制上面对于CalculateRewardPoints方法的期望，只是每个账户类型的测试目标系统不一样。

白金卡和金卡类型的奖励积分算法也与账户当前余额有关，上面示例的代码也显示了这一点，因此这些类型都可以单独进行编译。Balance属性现在可以公开设置了，这样单元测试也会与前面的不一样。图4-8中的UML类图展示了这些类之间的关联。

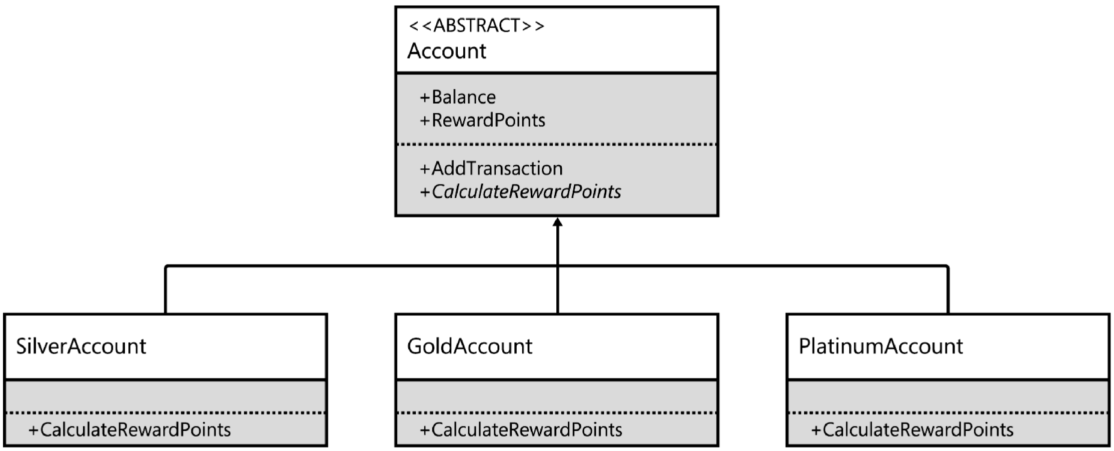


图4-8 因为有CalculateRewardPoints这个抽象方法，Account也变成了抽象类

这一步是实现完整替换switch语句目标过程中的一个小目标。每次不要做太多也很重要，因为只有这样你才能够确保自己的每个小改动是成功的。下一步要做的改动是把四个类型组织到一个继承层级结构下，如代码清单4-31所示。

代码清单4-31　Account类中最复杂的部分已经被移除了

```
public abstract class AccountBase
{
    public decimal Balance
    {
        get;
        private set;
    }

    public int RewardPoints
    {
        get;
        private set;
    }

    public void AddTransaction(decimal amount)
    {
        RewardPoints += CalculateRewardPoints(amount);
        Balance += amount;
    }

    public abstract int CalculateRewardPoints(decimal amount);
}
```

没有了switch语句，type数据也就没有必要存在了，因此带有参数的构造函数也不需要了。抽象的计算方法导致这个类也要变成抽象的，这就意味着，你不能实例化它了，当然也不能再直接测试它了。

单元测试需要一个对象实例才可以工作，因此下一步就是把这三个账户类型作为这个基类的子类。给抽象类命名添加后缀Base与接口命名前有前缀I一样，都是很有用的命名约定。从后缀Base上可以直接看出该类无法实例化且有关联的子类。

现在，你可以从GoldAccount类和PlatinumAccount类中移除Balance属性了，因为它们可以直接继承基类的Balance属性和AddTransaction方法成员。代码清单4-32展示了这次重构后的代码。

代码清单4-32　完成继承基类重构后的代码

```
public class SilverAccount : AccountBase
{
    public override int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Floor(amount / SilverTransactionCostPerPoint), 0);
    }

    private const int SilverTransactionCostPerPoint = 10;
}
// . . .
public class GoldAccount : AccountBase
{
```

```
    public override int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Floor((Balance / GoldBalanceCostPerPoint) + (amount /
  GoldTransactionCostPerPoint)), 0);
    }

    private const int GoldTransactionCostPerPoint = 5;
    private const int GoldBalanceCostPerPoint = 2000;
}
// . . .
public class PlatinumAccount : AccountBase
{
    public override int CalculateRewardPoints(decimal amount)
    {
        return Math.Max((int)decimal.Ceiling((Balance / PlatinumBalanceCostPerPoint) +
(amount / PlatinumTransactionCostPerPoint)), 0);
    }

    private const int PlatinumTransactionCostPerPoint = 2;
    private const int PlatinumBalanceCostPerPoint = 1000;
}
```


至此，这个重构就完成了。从此开始，为支持新增加的账户类型只需要创建一个新的AccountBase类的子类并提供自己的CalculateRewardPoints方法实现即可。无需改变任何已有代码，你需要做的只是针对新的奖励积分算法再编写一些单元测试。

3. 用工厂方法替代构造函数

在改善Account类的过程中，你会发现上面的几个重构会给其他一些地方带来副作用。重构前，使用该类的客户端使用Account类的带有账户类型参数的构造函数来创建账户对象，重构后，客户端又如何创建想要的账户子类对象呢？

AccountType枚举可以用作AccountBase类的工厂方法的参数。尽管构造函数和new运算符一起使用可以返回该类的实例对象，而工厂方法能返回同一个继承层次结构下的很多种不同类型的对象。代码清单4-33展示了基类中实现的工厂方法。

代码清单4-33 switch语句又回来了，但是更加简洁了

```
public abstract class AccountBase
{
    public static AccountBase CreateAccount(AccountType type)
    {
        AccountBase account = null;
        switch(type)
        {
            case AccountType.Silver:
                account = new SilverAccount();
                break;
            case AccountType.Gold:
                account = new GoldAccount();
                break;
            case AccountType.Platinum:
```

```
                account = new PlatinumAccount();
                break;
        }
        return account;
    }
}
```

工厂方法能够减轻客户端代码的负担的两个关键点是：第一，它是静态方法，这就意味着客户端可以直接通过类调用它，而不需要类实例；第二，返回类是基类，这样做可以对客户端隐藏具体的子类账户。实际上，将子类声明为internal就可以确保程序集外不可见。这样客户端也无法直接构造子类的实例对象，也就消除了潜在的new代码味道。代码清单4-34比较了重构前后客户端和账户交互的方式。

尽管仍然有switch语句存在，但它变得更简洁了，同时也利用到了前面多形性重构的好处。

代码清单4-34 AccountService类在重构前后创建新账户方式的比较

```
public void CreateAccount(AccountType accountType)
{
    var newAccount = new Account(accountType);
    accountRepository.NewAccount(newAccount);
}
// . . .
public void CreateAccount(AccountType accountType)
{
    var newAccount = AccountBase.CreateAccount(accountType);
    accountRepository.NewAccount(newAccount);
}
```

上面的代码示例中比较了客户端（这里是指AccountService类）在重构前后创建新的账户的两种不同的方式。代码区别并不大，但是要注意到new运算符已经被静态的工厂方法替代了。这种方式很常见，它用适应能力更高的代码替代了非常具体的代码。与总是返回同一类实例的方法相比，工厂方法提供了更多的可能性，因为它们能够返回同一个基类下的所有子类实例。

用心的读者可能会发现静态工厂方法并不是最优的选择，因为它是一个天钩而不是一个塔吊，因此会影响到代码的可测性和适应能力。更好的选择应该是将CreateAccount方法安排在一个合适的接口里，下一节会详细讲解这种方法。

4 用工厂类替代构造函数

相对于工厂方法，更好的一种选择是工厂类。实际上，你不需要让用户总是使用单个工厂方法的实现，只需要给他们一个接口，如代码清单4-35所示。

代码清单4-35 IAccountFactory接口隐藏了创建账户实例的实现细节

```
public interface IAccountFactory
{
    AccountBase CreateAccount(AccountType accountType);
}
```

接口的方法实际上与工厂方法一样，只是它是个实例方法。接口的实现可以与前面静态工厂方法完全一样，这就意味着该实现要清楚地知道所有不同的账户类型。这样重构后，AccountService类和其他客户端都需要将该接口作为构造函数的一个参数，如代码清单4-36所示。

代码清单4-36　AccountService类使用构造函数的工厂实例参数来创建账户实例

```
public class AccountService
{
    public AccountService(IAccountFactory accountFactory, IAccountRepository
  accountRepository)
    {
        this.accountFactory = accountFactory;
        this.accountRepository = accountRepository;
    }

    public void CreateAccount(AccountType accountType)
    {
        var newAccount = accountFactory.CreateAccount(accountType);
        accountRepository.NewAccount(newAccount);
    }

    private readonly IAccountRepository accountRepository;
    private readonly IAccountFactory accountFactory;
}
```


现在这个AccountService类看起来才像它应有的样子了：通过组合使用多个合适粒度的接口来为用户界面层提供更大的目标。这里为了简短和明晰，在构造函数中略去了一些防止出现空引用的防卫语句，以及在CreateAccount方法中略去了一些用于封装域异常到服务异常中的try/catch代码块。

4.2.2　一个新的账户类型

此时，你能有足够的信心以最小的代码改动来实现新账户类型的需求吗？这个还是说不准。一种情况下，你可以小心翼翼地添加新的账户类型，但另外的情况下，你会发现现有的模型设计做了一些错误的假设，这些错误的假设就形成了技术债务。

1. 一个新的奖励账户

假设你的客户端都想要增加另外一种新的账户类型：青铜卡，它能得到银卡账户一半的奖励积分。为了支持这个新类型，只需要在域层进行两处改动。第一，你需要基于AccountBase基类创建新的子类，如代码清单4-37所示。

代码清单4-37　青铜卡账户是一种新的账户类型

```
internal class BronzeAccount : AccountBase
{
    public override int CalculateRewardPoints(decimal amount)
    {
```

```
        return Math.Max((int)decimal.Floor(amount / BronzeTransactionCostPerPoint), 0);
    }

    private const int BronzeTransactionCostPerPoint = 20;
}
```

这个简单改变包括一个新的验证你的期望的单元测试，以及一个新的包含计算奖励积分算法的类型。

无论是工厂类还是工厂方法，为了支持新的账户类型，你都需要改变它，以及定义可能账户类型的枚举。代码清单4-38展示了如何修改工厂类来支持新的青铜卡账户。

代码清单4-38　在switch语句下增加的新case是用来创建青铜卡账户的

```
public AccountBase CreateAccount(AccountType accountType)
{
    AccountBase account = null;
    switch (accountType)
    {
        case AccountType.Bronze:
            account = new BronzeAccount();
            break;
        case AccountType.Silver:
            account = new SilverAccount();
            break;
        case AccountType.Gold:
            account = new GoldAccount();
            break;
        case AccountType.Platinum:
            account = new PlatinumAccount();
            break;
    }
    return account;
}
```

在继续支持客户端需要的下一个新账户类型之前，你是否还能对该方法再做重构从而无需为每个新增类型修改方法代码？你不可以再使用在“用多态替代条件表达式”部分中讲述的重构方法了，因为工厂方法就是这种重构的结果。相反，是否可以从accountType名称构造AccountBase类的实例且无需直接引用每个名称的具体值和子类？代码清单4-39给出了答案。

代码清单4-39　如果所有账户都遵守一个确定的命名规则，这个工厂就足以处理它们了

```
public AccountBase CreateAccount(string accountType)
{
    var objectHandle = Activator.CreateInstance(null, string.Format("{0}Account",
  accountType));
    return (AccountBase)objectHandle.Unwrap();
}
```

注意看，这里采用了更加灵活的字符串值来代替枚举值。这也许会引入问题，因为任何字符串值都是可以接受的，而不是预先定义的可以匹配的有效账户类型。当然，这也是当前练习的要点。

这种重构有一点激进，因为它存在创建有漏洞的工厂抽象的风险，也就是说该工厂并不能响应所有的请求。应用这种重构需要先满足以下几个前提条件。

- 每个账户类型必须遵守命名规则[Type]Account，前缀[Type]是账户类型的枚举值。
- 每个账户类型必须与该工厂方法在同一个程序集中。
- 每个账户类型必须要有公共的默认构造函数（这种类型无法使用任何值进行参数化）。

这些限制条件通常意味着重构有些过度了，如果后面无法满足其中某个条件，这种重构就会引入问题。因此，请谨慎行事。

2. 代码味道：拒绝继承

在发起新的积分奖励活动后的某一天，客户的市场部门想知道参加活动的每种账户类型都有多少用户。你告诉他们整个积分奖励活动有个规则：单个用户只有一张卡，只能是青铜卡、银卡、金卡或白金卡中的一种。但是实际情况并不是这样的。因为没有考虑到不参与积分奖励活动的账户，而每个人默认拿到的都是青铜卡。这样分析后的结果就是，需要支持另外一种新的账户类型：标准账户。

标准账户与其他账户的目的不同，它不会得到任何积分奖励。要支持这个新的账户类型，有两种方法可以选择。第一，你可以创建一个新的AccountBase类的子类，如代码清单4-40所示，其中的CalculateRewardPoints方法会直接返回零，用来直接表达并不累计任何积分。

代码清单4-40 一种不计算任何积分奖励的简单账户类型

```
internal class StandardAccount : AccountBase
{
    public override int CalculateRewardPoints(decimal amount)
    {
        return 0;
    }
}
```

另外一种方式就是承认并不是所有账户都可以得到奖励积分，并且在实际的域模型中有两种不同的类型。在这种情况下，不要为CalculateRewardPoint方法提供默认实现，因为子类会拒绝继承父类的现有实现，这也是著名的代码味道“拒绝继承”。代码清单4-40的示例中，StandardAccount类直接拒绝实现接口的方法而不是忽略它，而接下来的重构方式将会拒绝实现整个接口。

3. 用委托替代继承

这种重构方式需要你把AccountBase类分为两个部分。接口的某些部分被挪到一个新的类型层次结构中来表示积分奖励卡，其余的继续留在AccountBase类中。这种方式通过委托积分奖励卡来替代账户类型的继承关系。

首先要做的改动是引入新的IRewardCard接口来定义积分奖励卡的属性和行为，如代码清单4-41所示。

代码清单4-41　积分奖励及其计算方法都从Account类中移走了

```
public interface IRewardCard
{
    int RewardPoints
    {
        get;
    }

    void CalculateRewardPoints(decimal transactionAmount, decimal accountBalance);
}
```

前面的实例中，这两个成员都属于AccountBase类，将它们移出是因为它们都取决于是否存在积分奖励卡。注意到接口中的CalculateRewardPoints方法有两处改动。第一处，该方法不再返回任何值，因为它希望的是直接修改RewardPoints属性。第二，你必须给该方法传入账户余额值，因为IRewardCard并不知道账户余额。这也是以这种方式分开两个对象带来的明显副作用：需要处理所有积分奖励卡中并没有封装的上下文数据。也许将来需要改变该方法的签名才可以消除这种副作用。

代码清单4-42展示了重构后的青铜卡和白金卡的IRewardCard接口的实现。

代码清单4-42　积分奖励卡实现的示例

```
internal class BronzeRewardCard : IRewardCard
{
    public int RewardPoints
    {
        get;
        private set;
    }

    public void CalculateRewardPoints(decimal transactionAmount, decimal accountBalance)
    {
        RewardPoints += Math.Max((int)decimal.Floor(transactionAmount /
  BronzeTransactionCostPerPoint), 0);
    }
    private const int BronzeTransactionCostPerPoint = 20;
}
// . . .
internal class PlatinumRewardCard : IRewardCard
{
    public int RewardPoints
    {
        get;
        private set;
    }
```

```
    public void CalculateRewardPoints(decimal transactionAmount, decimal accountBalance)
    {
        RewardPoints += Math.Max((int)decimal.Ceiling((accountBalance /
PlatinumBalanceCostPerPoint) + (transactionAmount / PlatinumTransactionCostPerPoint)), 0);
    }

    private const int PlatinumTransactionCostPerPoint = 2;
    private const int PlatinumBalanceCostPerPoint = 1000;
}
```

这些类与它们的Account类前身很类似，只是多了一个本地的RewardPoints属性。

如代码清单4-43所示，Account类不再是抽象的，也不再需要有Base后缀了。构造实例时，它会接受一个IRewardCard接口的实例并委托它在交易发生时计算奖励积分。整体来看，这个账户类型很像无需处理积分奖励需求时候的初始定义。

代码清单4-43　现在每个账户中都包含了一个积分奖励卡

```
public class Account
{
    public Account(IRewardCard rewardCard)
    {
        this.rewardCard = rewardCard;
    }

    public decimal Balance
    {
        get;
        private set;
    }

    public void AddTransaction(decimal amount)
    {
        rewardCard.CalculateRewardPoints(amount, Balance);
        Balance += amount;
    }

    private readonly IRewardCard rewardCard;
}
```

为了支持标准账户（没有积分奖励卡的账户），你要么给构造函数传入积分奖励卡的null引用（并且在引用前增加防卫语句来避免NullReferenceException异常），要么你直接定义一个NullRewardCard。后者也是应用空对象模式的一种实现，它在CalcaulateReardPoints方法被调用时不会累计任何积分奖励。

4.3　总结

本章是单元测试和重构的混合体，因为这二者总是同时出现并交替应用的。

每个单元测试都应该表达产品代码的期望，理想情况下，一个外行应该能理解这个期望。尽管测试和代码一样都是技术工件，单元测试用一组对象来验证对真实世界期望的行为，就像这些对象封装了真实世界的概念一样。

当你认真应用测试先行的方法时，你首先要编写的是一个失败的单元测试而不是产品代码。然后你再尝试编写最简单的产品代码来让单元测试由红色失败状态转为绿色成功状态。按照这种逻辑，产品代码就成为了实现单元测试期望行为的自然产出物。

当你在对代码做单元测试时，就为自己后期改动产品代码以使其具备更高的适应需求变更能力提供了一个坚实的基础。对代码的重构是一个渐进地改善已有代码设计的过程。具体的重构方式有很多种，本章只讲解了一部分。每种重构方式都有可能表示对某个域的折中以对另外一个域做出一定的改善，而且重构过程与编程的其他很多方面一样，都是很主观的。

到此，本书的敏捷基础这一部分就结束了。接下来的第二部分将会集中讲解SOLID代码以及它如何帮助你提高代码的自适应能力。

Part 2 第二部分

编写 SOLID 代码

本部分内容

SOLID 是一组最佳编码实践的首字母缩写。同时应用这些最佳实践，能提高代码适应变更的能力。尽管 Bob Martin 已经于 15 年前识别并引入了这组 SOLID 实践活动，它们应该被大家熟知，但是事实上它们并没有得到广泛传播。

这一部分中，每章都会专门讲解一个 SOLID 原则。

- **S**　单一职责原则
- **O**　开放与封闭原则
- **L**　Liskov 替换原则
- **I**　接口分离原则
- **D**　依赖注入原则

即便分开讲述这些原则，它们每一个都是值得所有软件开发人员认真学习的。如果配合使用这些原则，它们能给代码带来完全不同的结构，这种结构具备很强的适应变更的能力。

尽管如此，与其他模式和实践一样，它们也只是工具。如何选择在正确的时机和场合中应用模式或实践本身就是软件开发艺术的一部分。过度使用虽然可以让代码有很高的自适应能力，但会导致层次粒度过小而难以理解或使用。过度使用还会影响到代码质量的另一个关键因素：可读性。现代软件产品的开发已不再是一个人单打独斗，通常都是由一个团队协作完成。因此，审慎地决定模式、实践或 SOLID 原则的应用时机和场合是非常重要的，只有这样才可以保证代码在将来一直都是可以被完全理解的。

第 5 章

单一职责原则

完成本章学习之后，你将学到以下技能。

- 理解单一职责原则的重要性。
- 识别出职责太多的类。
- 编写具有单一职责的方法、类和模块。
- 将巨型类重构为多个具有单一职责的小型类。
- 使用设计模式来分离职责。

单一职责原则（Single Responsibility Principle，SPR）要求开发人员所编写的代码有且只有一个变更理由。如果一个类有多个变更理由，那么它就具有多个职责。多职责类应该被分解为多个单职责类。

本章会讲解如何创建有用的单职责类。通过委托和抽象，包含多个变更理由的类应该把一个或多个职责委托给其他的单职责类。

抽象的重要性再强调也不为过。它是自适应代码的支撑，没有它，开发人员就只能疲于奔命地应付Scrum和其他敏捷流程积极响应的变更需求了。

5.1 问题描述

为了更好地讲解拥有太多职责类的问题，本节会通过一个示例做详细的剖析。代码清单5-1展示了一个简单的交易处理器类，它能从文件读取记录并更新数据库。尽管它现在看起来还很小，但为了满足一些需求，你需要在此基础上持续添加新特性。

代码清单5-1 一个拥有太多职责类的示例

```
public class TradeProcessor
{
    public void ProcessTrades(System.IO.Stream stream)
    {
        // read rows
        var lines = new List<string>();
        using(var reader = new System.IO.StreamReader(stream))
        {
            string line;
            while((line = reader.ReadLine()) != null)
```

```
            {
                lines.Add(line);
            }
        }

        var trades = new List<TradeRecord>();

        var lineCount = 1;
        foreach(var line in lines)
        {
            var fields = line.Split(new char[] { ',' });

            if(fields.Length != 3)
            {
                Console.WriteLine("WARN: Line {0} malformed. Only {1} field(s) found.",
    lineCount, fields.Length);
                continue;
            }

            if(fields[0].Length != 6)
            {
                Console.WriteLine("WARN: Trade currencies on line {0} malformed: '{1}'",
    lineCount, fields[0]);
                continue;
            }

            int tradeAmount;
            if(!int.TryParse(fields[1], out tradeAmount))
            {
                Console.WriteLine("WARN: Trade amount on line {0} not a valid integer:
    '{1}'", lineCount, fields[1]);
            }

            decimal tradePrice;
            if (!decimal.TryParse(fields[2], out tradePrice))
            {
                Console.WriteLine("WARN: Trade price on line {0} not a valid decimal:
    '{1}'", lineCount, fields[2]);
            }

            var sourceCurrencyCode = fields[0].Substring(0, 3);
            var destinationCurrencyCode = fields[0].Substring(3, 3);

            // calculate values
            var trade = new TradeRecord
            {
                SourceCurrency = sourceCurrencyCode,
                DestinationCurrency = destinationCurrencyCode,
                Lots = tradeAmount / LotSize,
                Price = tradePrice
            };

            trades.Add(trade);

            lineCount++;
```

```
        }

        using (var connection = new System.Data.SqlClient.SqlConnection("Data
Source=(local);Initial Catalog=TradeDatabase;Integrated Security=True"))
        {
            connection.Open();
            using (var transaction = connection.BeginTransaction())
            {
                foreach(var trade in trades)
                {
                    var command = connection.CreateCommand();
                    command.Transaction = transaction;
                    command.CommandType = System.Data.CommandType.StoredProcedure;
                    command.CommandText = "dbo.insert_trade";
                    command.Parameters.AddWithValue("@sourceCurrency", trade.
SourceCurrency);
                    command.Parameters.AddWithValue("@destinationCurrency", trade.
DestinationCurrency);
                    command.Parameters.AddWithValue("@lots", trade.Lots);
                    command.Parameters.AddWithValue("@price", trade.Price);

                    command.ExecuteNonQuery();
                }

                transaction.Commit();
            }
            connection.Close();
        }

        Console.WriteLine("INFO: {0} trades processed", trades.Count);
    }

    private static float LotSize = 100000f;
}
```

这不仅仅是一个单个类拥有太多职责的示例，也是一个单一方法拥有太多职责的示例。在仔细阅读代码后，你才能知道这个类在尝试达成以下目标。

(1) 从一个Stream参数中读出每行内容并存放到一个字符串列表中。

(2) 解析出每行内容中的一组数据并把它们存放在一个更结构化的TradeRecord实例的列表中。

(3) 整个分析过程中包括了一些校验数据的动作和将日志输出到控制台的动作。

(4) 枚举每个TradeRecord实例，并调用了一个存储子流程来将数据存放到一个数据库中。

TradeProcessor的职责包括：读取流数据，解析字符串，验证数据，记录日志，以及向数据库插入数据。单一职责原则要求这个类和其他类一样应该只有一个变更理由。然而，TradeProcessor的现状则是会在以下场合都会发生变更。

- 当你决定用远程Web服务来代替Stream作为输入源时。
- 当输入数据的格式变化时，也许会增加一个新的数据项来表示交易的代理人。
- 当输入数据的验证规则发生变化时。

- 当你输出警告、错误和信息日志的方式改变时。输出给控制台的方式对远程Web服务来说是不可行的。
- 当数据库也发生了某些变化时，也许是insert_trade存储过程也需要一个额外的代理人参数，或者你决定使用文档存储来代替关系型数据库时，又或者将数据库移到你所使用的Web服务后台时。

这些改变必然需要修改该类的实现。更进一步说，除非你维护了很多种版本，否则你不可能让TradeProcessor能够同时适配多种情况，比如不同的数据输入源。想象一下，在只有几个命令行参数的情况下，如果要求将交易数据存储在Web服务上，你会有多头疼啊。

5.1.1　重构清晰度

将TradeProcessor重构为单职责类的第一步就是把ProcessTrades方法拆分为多个更小的方法，每个方法专注完成一个职责。重构产生的每个方法都会在接下来的代码清单中进行展示，代码清单后还会讲解改动的细节。

代码清单5-2展示了重构后的ProcessTrades方法，它现在只是委托其他几个方法做事。

代码清单5-2　因为将工作委托给了其他方法，所以ProcessTrades方法现在最小

```
public void ProcessTrades(System.IO.Stream stream)
{
    var lines = ReadTradeData(stream);
    var trades = ParseTrades(lines);
    StoreTrades(trades);
}
```

原始的ProcessTrades方法代码可以分为三个部分：从流中读取交易数据，将字符串数据转换为TradeRecord实例，以及将交易数据写入永久存储中。注意，方法的输出会成为下一个方法的输入。如果没有从ParseTrades方法中返回的交易记录数据，你就无法调用StoreTrades方法，同样，直到ReadTradeData方法返回字符串后，你才可以调用ParseTrades方法。

按照顺序，我们先看看ReadTradeData方法，如代码清单5-3所示。

代码清单5-3　ReadTradeData封装了原有的代码

```
private IEnumerable<string> ReadTradeData(System.IO.Stream stream)
{
    var tradeData = new List<string>();
    using (var reader = new System.IO.StreamReader(stream))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            tradeData.Add(line);
        }
    }
    return tradeData;
}
```

这个方法里面的代码直接从原有的ProcessTrades方法中提取。经过简单的封装后，它用一个字符串枚举返回了读取到的所有字符串数据。注意，此处与原始代码是有所不同的，返回的字符串枚举是只读的，而原有实现中却无法阻止后续代码添加更多的字符串。

接下来的代码清单5-4展示的是ParseTrades方法。它的代码与原有实现不完全相同，因为它自己也要委托一些任务给其他方法。

代码清单5-4　ParseTrades通过委托其他方法来减小自己代码的复杂度

```
private IEnumerable<TradeRecord> ParseTrades(IEnumerable<string> tradeData)
{
    var trades = new List<TradeRecord>();
    var lineCount = 1;
    foreach (var line in tradeData)
    {
        var fields = line.Split(new char[] { ',' });

        if(!ValidateTradeData(fields, lineCount))
        {
            continue;
        }

        var trade = MapTradeDataToTradeRecord(fields);

        trades.Add(trade);

        lineCount++;
    }
    return trades;
}
```

该方法把数据校验和映射这两个职责委托给了其他两个方法。如果不做这种委托，这部分的处理逻辑依然会因带有太多职责从而太过复杂。代码清单5-5会展示ValidateTradeData方法，它返回一个布尔值来表明交易记录数据是否有效。

代码清单5-5　所有的校验代码都在这一个方法内

```
private bool ValidateTradeData(string[] fields, int currentLine)
{
    if (fields.Length != 3)
    {
        LogMessage("WARN: Line {0} malformed. Only {1} field(s) found.", currentLine,
  fields.Length);
        return false;
    }

    if (fields[0].Length != 6)
    {
        LogMessage("WARN: Trade currencies on line {0} malformed: '{1}'", currentLine,
  fields[0]);
        return false;
```

```
        }

        int tradeAmount;
        if (!int.TryParse(fields[1], out tradeAmount))
        {
            LogMessage("WARN: Trade amount on line {0} not a valid integer: '{1}'",
      currentLine, fields[1]);
            return false;
        }

        decimal tradePrice;
        if (!decimal.TryParse(fields[2], out tradePrice))
        {
            LogMessage("WARN: Trade price on line {0} not a valid decimal: '{1}'",
      currentLine, fields[2]);
            return false;
        }

        return true;
    }
```

相对于原始代码，唯一的改动就是委托另外一个方法来记录日志。代码清单5-6展示了这个改动，使用LogMessage方法而不是直接调用Console.WriteLine。

代码清单5-6　LogMessage方法只相当于给Console.WriteLine起了一个别名

```
    private void LogMessage(string message, params object[] args)
    {
        Console.WriteLine(message, args);
    }
```

代码清单5-7展示了ParseTrades方法委托的另外一个方法。它将一组从流中读取到的字符串映射为一组TradeRecord类的实例。

代码清单5-7　从一个类型映射到另外一个类型是一个独立的职责

```
    private TradeRecord MapTradeDataToTradeRecord(string[] fields)
    {
        var sourceCurrencyCode = fields[0].Substring(0, 3);
        var destinationCurrencyCode = fields[0].Substring(3, 3);
        var tradeAmount = int.Parse(fields[1]);
        var tradePrice = decimal.Parse(fields[2]);

        var tradeRecord = new TradeRecord
        {
            SourceCurrency = sourceCurrencyCode,
            DestinationCurrency = destinationCurrencyCode,
            Lots = tradeAmount / LotSize,
            Price = tradePrice
        };

        return tradeRecord;
    }
```

重构生成的第六个也是最后一个方法是StoreTrades，如代码清单5-8所示。这个方法封装了与数据库交互的代码。它也委托了前面讲到的LogMessage方法来记录信息日志。

代码清单5-8 在最后的StoreTrades方法就绪后，该类的各个职责已经清楚地划分开了

```
private void StoreTrades(IEnumerable<TradeRecord> trades)
{
    using (var connection = new System.Data.SqlClient.SqlConnection("Data
  Source=(local);Initial Catalog=TradeDatabase;Integrated Security=True"))
    {
        connection.Open();
        using (var transaction = connection.BeginTransaction())
        {
            foreach (var trade in trades)
            {
                var command = connection.CreateCommand();
                command.Transaction = transaction;
                command.CommandType = System.Data.CommandType.StoredProcedure;
                command.CommandText = "dbo.insert_trade";
                command.Parameters.AddWithValue("@sourceCurrency", trade.SourceCurrency);
                command.Parameters.AddWithValue("@destinationCurrency",
  trade.DestinationCurrency);
                command.Parameters.AddWithValue("@lots", trade.Lots);
                command.Parameters.AddWithValue("@price", trade.Price);

                command.ExecuteNonQuery();
            }

            transaction.Commit();
        }
        connection.Close();
    }

    LogMessage("INFO: {0} trades processed", trades.Count());
}
```

回顾一下刚刚完成的重构，相对于原有的实现而言已经有了明显的改进。然而，你真的得到什么了吗？尽管新的ProcessTrades方法比原来的巨型方法小了很多，代码也更加容易阅读了，但是你并没有将代码的自适应能力提高多少。比如，你现在能改动LogMessage方法的实现，用文件替代控制台作为日志输出的目标，但这会涉及你不想看到的对TradeProcessor类的改动。

以上重构为最终实现划分该类职责的目标奠定了坚实的基础。此次重构的目标是清晰度，而不是自适应能力。接下来的任务就是将每个职责划分到不同的类中，并把它们隐藏在相应的接口后面。现在你需要做真正的抽象，因为这样才能真正提高代码的自适应能力。

5.1.2 重构抽象

构建于新的TradeProcessor实现之上，接下来的重构会引入多个新的抽象，它们几乎能满足该类上所有的变更请求。尽管这个示例看起来似乎很小，甚至微不足道，但对于当前这个主题

的教程来说它仍是一个很合适的范例。应用程序规模从小到大逐步发展的情况也很普遍。再小的程序，只要有一些人开始使用，新的特性需求就会蜂拥而至。

通常，术语原型（prototype）和概念验证（proof of concept）会用于描述这种小的应用程序，而从原型到产品应用程序的转变几乎是无缝的。这也是为什么重构抽象的能力能够作为自适应开发检验标准的原因。如果不做抽象重构，大量的只带有模糊职责和抽象定义的需求就会发展成为一个“大泥球”：一个包含一个或一组类的程序集。最终得到的应用程序没有相应的单元测试，又很难维护和增强，而它还可能是业务链上一个很重要的节点。

重构TradeProcessor抽象的第一步就是设计一个或一组接口来执行三个最高级别的任务：读取数据，处理数据和存储数据。图5-1展示了第一组抽象。

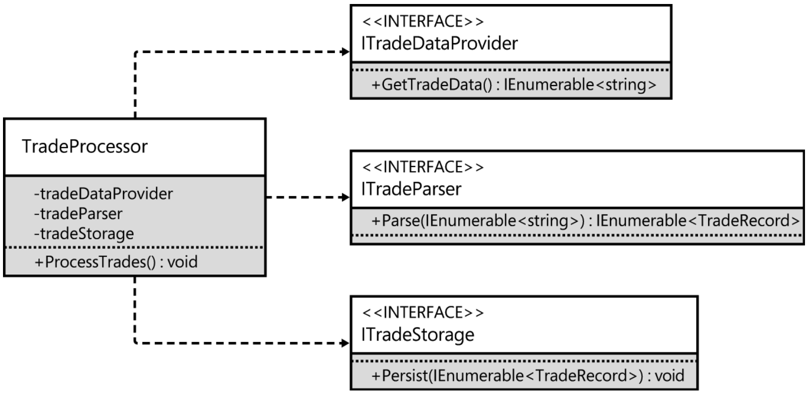


图5-1　TradeProcessor将会依赖三个新的接口

基于上一节重构分割ProcessTrades方法代码形成的三个方法，你应该清楚如何开始第一组抽象。根据单一职责原则的定义，这三个主要的职责应该由不同的类来负责。此外，前面几章也提到过，你不应该让一个类直接依赖其他类，而应该通过接口。因此，这三个主要职责会被提取到三个独立的接口中。代码清单5-9展示了这样重构后的TradeProcessor类。

代码清单5-9　现在TradeProcessor只是简单地封装了一个流程

```
public class TradeProcessor
{
    public TradeProcessor(ITradeDataProvider tradeDataProvider, ITradeParser tradeParser,
  ITradeStorage tradeStorage)
    {
        this.tradeDataProvider = tradeDataProvider;
        this.tradeParser = tradeParser;
        this.tradeStorage = tradeStorage;
    }

    public void ProcessTrades()
    {
        var lines = tradeDataProvider.GetTradeData();
```

```
            var trades = tradeParser.Parse(lines);
            tradeStorage.Persist(trades);
        }

        private readonly ITradeDataProvider tradeDataProvider;
        private readonly ITradeParser tradeParser;
        private readonly ITradeStorage tradeStorage;
    }
```

现在TradeProcessor类已经与原始的实现有了天壤之别。它现在不包括任何交易处理流程的细节实现，取而代之的是整个流程的蓝图。这个类现在只对交易数据格式转换的流程建模，这是它的唯一职责，也是引起该类后续变更的唯一原因。如果流程本身发生改变，该类也需要改变以反映更新后的流程。你如果只是打算不接受流数据，不再将日志输出到控制台，或者不再将交易数据存储到数据库中，这些都不会影响TradeProcessor类。

根据阶梯模式（在第2章中介绍过）的定义，TradeProcessor类依赖的所有接口都应该在各自独立的程序集内。这样就保证了TradeProcessor类的客户端或接口的实现程序集之间没有相互依赖。三个接口的实现类StreamTradeDataProvider、SimpleTradeParser和AdoNetTradeStorage可以分布在三个不同的程序集中。这三个类型有个共同的命名约定：用实现所需的具体上下文信息代替了接口名称的前缀I。因此StreamTradeProvider顾名思义就是从Stream获取数据的ITradeProvider接口的一个实现；AdoNetTradeStorage类就是指使用ADO.NET将交易数据存储到ITradeStorage接口的一个实现；而SimpleTradeParser类名中的Simple则表示该类没有其他上下文的依赖关系。

这三个实现类可以放置在同一个程序集中，因为它们都依赖Microsoft .NET Framework的一组核心程序集。如果要引入的实现依赖的是第三方程序集、第一方程序集或者非核心的.NET Framework程序集，你就应该把它们布置在各自独立的程序集中。比如，你打算用Dapper映射库替代ADO.NET，你会创建一个名为Services.Dapper的新程序集，其中包含了ITradeStorage接口的实现DapperTradeStorage类。

ITradeDataProvider接口并不依赖Stream类。而上一节中用来获取交易数据的方法需要一个Stream实例作为传入参数，但是这样做明显会让该方法依赖Stream所在的程序集。当你在创建接口并做抽象重构时，不要保留那些会对代码自适应能力有副作用的依赖关系，这一点很重要。我们已经知道还可以从Stream之外的数据源获取交易数据，因此重构后的接口已经不包括对Stream的依赖了。取而代之的是，StreamTradeProvider类需要一个Stream实例作为其构造函数的传入参数，而不是成员方法的传入参数。通过使用构造函数，你可以建立需要的任何依赖关系，而且不会影响接口。代码清单5-10展示了StreamTradeProvider的实现。

代码清单5-10 可以通过构造函数的传入参数将上下文传入方法，保持接口不受影响

```
    public class StreamTradeDataProvider : ITradeDataProvider
    {
        public StreamTradeDataProvider(Stream stream)
        {
```

```
        this.stream = stream;
    }

    public IEnumerable<string> GetTradeData()
    {
        var tradeData = new List<string>();
        using (var reader = new StreamReader(stream))
        {
            string line;
            while ((line = reader.ReadLine()) != null)
            {
                tradeData.Add(line);
            }
        }
        return tradeData;
    }

    private Stream stream;
}
```

记住，作为客户端的TradeProcessor类现在不清楚，当然也不应该清楚StreamTradeDataProvider类的任何实现细节，现在它只能通过ITradeDataProvider接口的GetTradeData方法来获取数据。

TradeProcessor类这个示例还可以提取更多的抽象。比如原有的ParseTrades方法被委托承担数据校验和类型映射的职责，你可以通过重复重构来实现只具备单一职责的SimpleTradeParser类。图5-2展示了重构得到的SimpleTradeParser类的UML图。

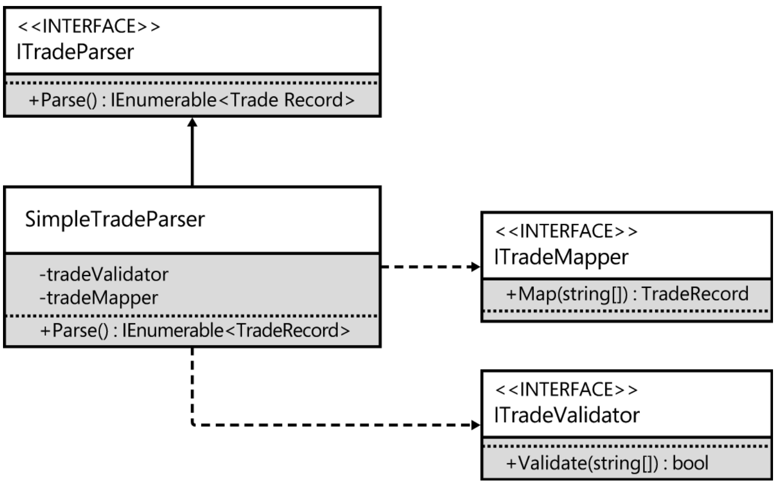


图5-2　重构SimpleTradeParser类后产生的新类也只具备单一职责

将职责抽象成为接口（以及相应的实现）的过程是递归的。在检视每个类时，你需要判断它是否具备多重职责，如果是，提取职责的抽象直到该类只具备单个职责。代码清单5-11展示的是

SimpleTradeParser类，它同样在需要的时候委托其他接口来辅助完成自己的任务。对于它而言，唯一的变更缘由就是交易数据整体结构的改变，比如，可能会用tab替代逗号来分隔字符串数据，或者用XML结构来代替简单结构的字符串。

代码清单5-11　解析交易数据的算法封装在ITradeParser接口的实现当中

```
public class SimpleTradeParser : ITradeParser
{
    public SimpleTradeParser(ITradeValidator tradeValidator, ITradeMapper tradeMapper)
    {
        this.tradeValidator = tradeValidator;
        this.tradeMapper = tradeMapper;
    }

    public IEnumerable<TradeRecord> Parse(IEnumerable<string> tradeData)
    {
        var trades = new List<TradeRecord>();
        var lineCount = 1;
        foreach (var line in tradeData)
        {
            var fields = line.Split(new char[] { ',' });

            if (!tradeValidator.Validate(fields))
            {
                continue;
            }

            var trade = tradeMapper.Map(fields);

            trades.Add(trade);

            lineCount++;
        }
        return trades;
    }

    private readonly ITradeValidator tradeValidator;
    private readonly ITradeMapper tradeMapper;
}
```

最后一个重构的目标是将日志功能的抽象从两个使用它的类中提取出来。现在ITrade-Validator和ITradeStorage两个接口的实现过程中都会直接将日志输出到控制台中。这次重构，你不需要再实现自己的日志类型，而是创建一个适配器类来调用流行的日志库Log4Net。图5-3中的UML图展示了这样重构后的结果。

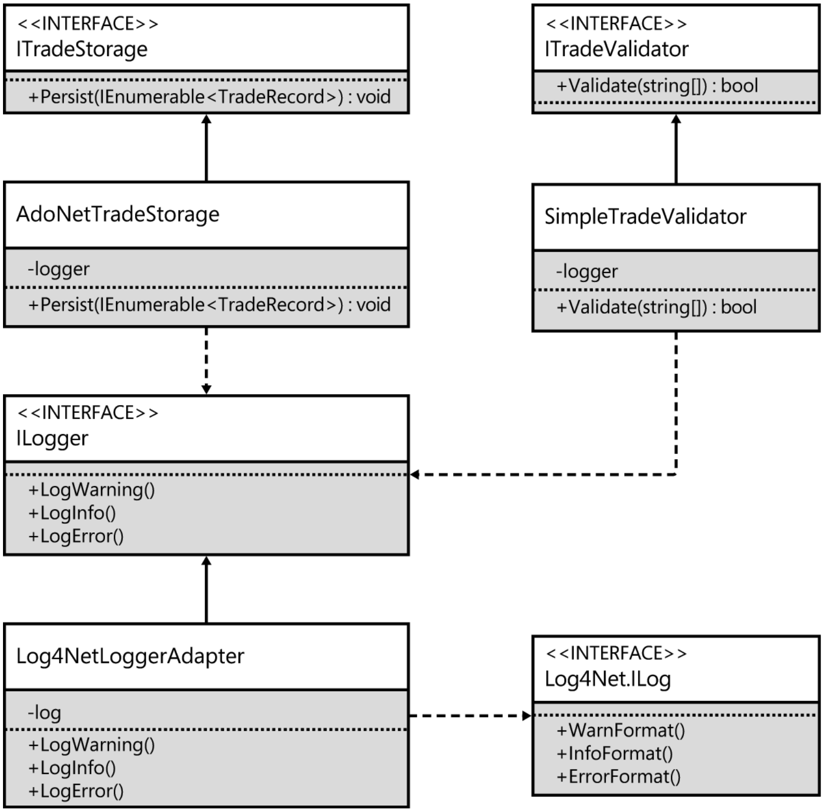


图5-3 通过为Log4Net实现一个适配器，就无需在每个程序集中都引用它了

创建诸如`Log4NetLoggerAdapter`等适配器类的好处是，你可以通过它们把第三方引用转换为第一方引用。注意看，`AdoNetTradeStorage`和`SimpleTradeValidator`两个类都依赖第一方的`ILogger`接口，但在运行时实际调用的依然是Log4Net的程序集。需要引用Log4Net程序集的地方只包括应用程序的入口（详见第9章）以及新创建的`Service.Log4Net`程序集。这样，所有需要依赖Log4Net的代码，比如定制的输出器等，也都应该位于`Service.Log4Net`程序集中。当前示例中，目前只有适配器类的实现在这个程序集中。

代码清单5-12展示了重构后的数据校验类型。它现在不再依赖控制台输出了。因为Log4Net的灵活实现，你实际可以在任何地方调用记录日志了。这里对于你所关心的日志问题，代码已经达到了完全自适应的程度。

代码清单5-12 重构后的`SimpleTradeValidator`类

```
public class SimpleTradeValidator : ITradeValidator
{
    private readonly ILogger logger;
    public SimpleTradeValidator(ILogger logger)
    {
```

```
            this.logger = logger;
        }

        public bool Validate(string[] tradeData)
        {
            if (tradeData.Length != 3)
            {
                logger.LogWarning("Line malformed. Only {1} field(s) found.",
      tradeData.Length);
                return false;
            }

            if (tradeData[0].Length != 6)
            {
                logger.LogWarning("Trade currencies malformed: '{1}'", tradeData[0]);
                return false;
            }

            int tradeAmount;
            if (!int.TryParse(tradeData[1], out tradeAmount))
            {
                logger.LogWarning("Trade amount not a valid integer: '{1}'", tradeData[1]);
                return false;
            }

            decimal tradePrice;
            if (!decimal.TryParse(tradeData[2], out tradePrice))
            {
                logger.LogWarning("WARN: Trade price not a valid decimal: '{1}'",
      tradeData[2]);
                return false;
            }

            return true;
        }
    }
```

到这里，我们需要一个快速的回顾。请记住，对于代码的功能你没有做任何改动，代码在功能上和原来是完全一样的。尽管如此，如果你就是想要增强功能，也可以很轻松地做到。为满足新需求而给代码增加新功能不只是扩展和重构现有代码，还需要增加实现新功能的新代码。

参考前面已经列举的潜在的代码增强点，重构后的新版本能在无需改变任何现有类的情况下实现以下需求的增强功能。

- 需求：当你决定用远程Web服务来代替Stream做输入源时。
  - 解决方案：创建一个ITradeDataProvider接口的新实现类来支持从服务获取数据。
- 需求：当输入数据的格式变化时，也许会增加了一个新的数据项来表示交易的代理人。
  - 解决方案：改变ITradeDataValidator、ITradeDataMapper以及ITradeStorage三个接口的实现以支持处理新的代理人数据。
- 需求：当输入数据的验证规则变化时。

- 解决方案：修改ITradeDataValidator接口的实现以反映最新的规则。

- 需求：当你输出警告、错误和信息日志的方式改变时。输出给控制台的方式对远程Web服务来说是不可行的。
    - 解决方案：已经讨论过了，通过适配器访问的Log4Net已经提供了非常丰富的日志记录方法了。
- 需求：当数据库也发生了某些变化时，也许是insert_trade存储过程也需要一个额外的代理人参数，或者你决定使用文档存储来代替关系型数据库，又或者将数据库移到你所使用的Web服务后台。
    - 解决方案：如果存储过程改变了，你需要编辑AdoNetTradeStorageI类来包含代理人数据的处理。对于其他两个改变，你需要创建MongoTradeStorage类来使用MongoDB存储交易数据，你还需要创建一个ServiceTradeStorage类来隐藏Web服务后的实现。

我希望现在你已经对使用接口进行抽象，按照阶梯模式解耦程序集，渐进地重构，以及坚持单职责原则这些方式很有信心了，合理地组合使用它们才能编写出自适应代码。

让代码清晰地委托抽象完成自身职责的方式还有很多种。本章剩余部分会集中讲解能让每个类都集中在单个职责上的其他方式。

5.2　单一职责原则和修饰器模式

修饰器模式（Decorator Pattern，DP）能够很好地确保每个类只有单个职责。一般情况下，完成很多事情的类并不能轻易地将职责划分到其他类型中，因为很多职责看起来是相互关联的。

修饰器模式的前置条件是：每个修饰器类实现一个接口且能同时接受一个或多个同一个接口实例作为构造函数的输入参数。这样做的好处是，可以给已经实现了某个特定接口的类添加功能，而且修饰器同时也是所需接口的一个实现，并且对用户不可见。图5-4展示了修饰器模式的UML图。

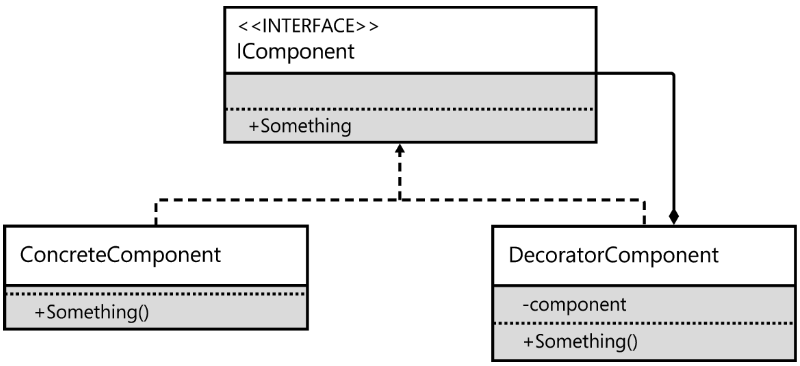


图5-4　修饰器模式实现的UML图

代码清单5-13展示了该模式的一个简单示例，它只是一个概念上的实例，无法实际使用。

代码清单5-13 修饰器模式的示例

```
public interface IComponent
{
    void Something();
}
// . . .
public class ConcreteComponent : IComponent
{
    public void Something()
    {

    }
}
// . . .
public class DecoratorComponent : IComponent
{
    public DecoratorComponent(IComponent decoratedComponent)
    {
        this.decoratedComponent = decoratedComponent;
    }

    public void Something()
    {
        SomethingElse();
        decoratedComponent.Something();
    }

    private void SomethingElse()
    {

    }

    private readonly IComponent decoratedComponent;
}
// . . .
class Program
{
    static IComponent component;

    static void Main(string[] args)
    {
        component = new DecoratorComponent(new ConcreteComponent());
        component.Something();
    }
}
```

上面的代码示例中，客户端从构造函数方法参数中接受了接口实例，你可以给用户提供原有的未修饰类的实例，也可以提供已修饰类的实例。注意，无论你提供的是未修饰的原始类还是已修饰的类，客户端都无需做任何改变。

5.2.1　复合模式

复合模式（Composite Pattern）是修饰器模式的一个特例，也是应用最广泛的修饰器模式。图5-5展示了复合模式的UML图。

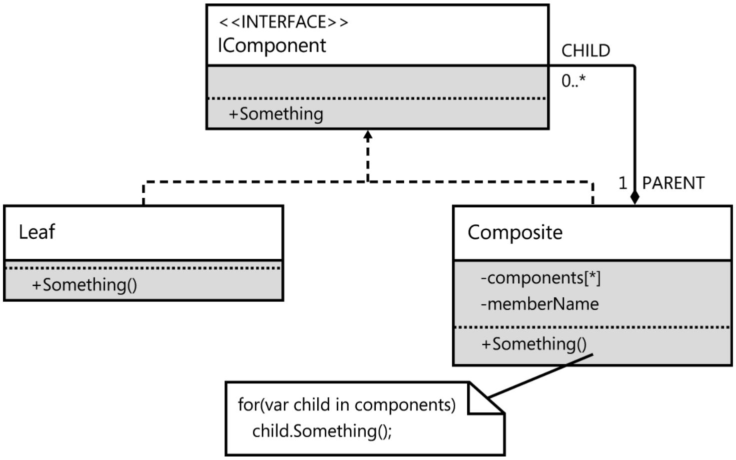


图5-5　复合模式和修饰器模式非常相似

复合模式的目的就是让你能把某个接口的一组实例看作该接口的一个实例。因此，客户端只需要接受接口的一个实例，在无需任何改变的情况下就能隐式地使用该接口的一组实例。代码清单5-14展示了一个实践中的复合修饰器。

代码清单5-14　一个接口的组合实现

```
public interface IComponent
{
    void Something();
}
// . . .
public class Leaf : IComponent
{
    public void Something()
    {

    }
}
// . . .
public class CompositeComponent : IComponent
{
    public CompositeComponent()
    {
        children = new List<IComponent>();
    }

    public void AddComponent(IComponent component)
```

```
        {
            children.Add(component);
        }

        public void RemoveComponent(IComponent component)
        {
            children.Remove(component);
        }

        public void Something()
        {
            foreach(var child in children)
            {
                child.Something();
            }
        }

        private ICollection<IComponent> children;
    }
    // . . .
    class Program
    {
        static void Main(string[] args)
        {
            var composite = new CompositeComponent();
            composite.AddComponent(new Leaf());
            composite.AddComponent(new Leaf());
            composite.AddComponent(new Leaf());

            component = composite;
            component.Something();
        }

        static IComponent component;
    }
```


CompositeComponent类中有增删IComponent接口实例的方法。这些方法并不是IComponent接口的组成部分，只被CompositeComponent类的客户端直接使用。无论哪个创建CompositeComponent类实例的工程方法或类型，也都需要能创建被修饰的实例并将它们传入到Add方法；否则，使用IComponent接口的客户端就必须为了配合组合而做改变。

无论客户端何时调用CompositeComponent类的Something方法，组合列表中的所有IComponent接口实例的Something方法都会被调用一次。这就是你将IComponent接口的单个实例（实现该接口的CompositeComponent类）的调用重新路由给该接口的很多实例（实现该接口的叶类）的方式。

每个你提供给CompositeComponent类的实例都必须实现IComponent接口（这是由编译器根据C#语言的强类型特性来保证的），但是这些实例不一定是同一种具体实现类。借助多态的强大能力，你能把所有该接口的实现类的实例只看作接口的实例。代码清单5-15展示了一个复合模式增强应用的示例，其中提供给CompositeComponent类的是多种不同的实现IComponent接口的

子类。

代码清单5-15　提供给组合列表的实例可以是不同的子类

```
public class SecondTypeOfLeaf : IComponent
{
    public void Something()
    {

    }
}
// . . .
public class AThirdLeafType : IComponent
{
    public void Something()
    {

    }
}
// . . .
public void AlternativeComposite()
{
    var composite = new CompositeComponent();
    composite.AddComponent(new Leaf());
    composite.AddComponent(new SecondTypeOfLeaf());
    composite.AddComponent(new AThirdLeafType());

    component = composite;
    composite.Something();
}
```

根据复合模式的逻辑设计，你甚至可以通过Add方法添加一个或多个CompositeComponent类的实例，这样就可以形成树状层次结构的一组实例链。

何时使用组合?

第2章介绍的随从反模式告诉我们，实现不应该与其接口位于相同的程序集中。然而，该规则有个例外情况：当实现的依赖是接口依赖的子集时。

有些组合的具体实现并不会引入更多的依赖，在这种情况下，组合类接口所在的程序集也可以同时包含组合类的具体实现。

第2章中，可以将类实例建模为对象图。这里继续使用对象图来进一步展示复合模式的工作方式。图5-6中，图中节点表示对象实例，有向边线则代表了方法调用。

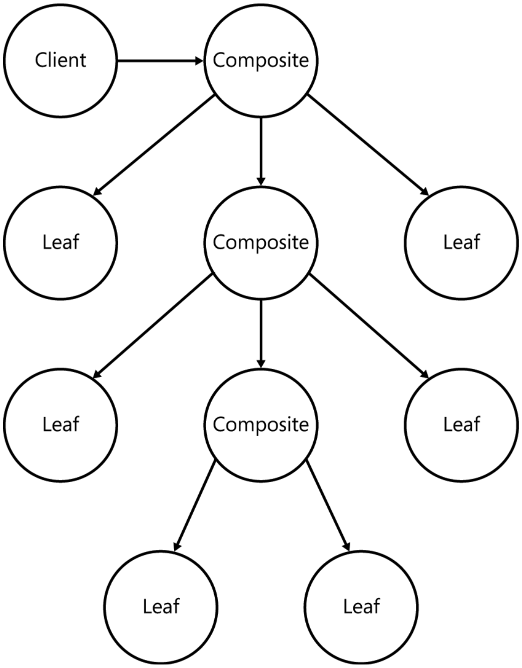


图5-6　对象图可以形象地展示程序的运行时结构

5.2.2　谓词修饰器

谓词修饰器（Predicate Decorator）能够很好地消除客户端代码中的条件执行语句。代码清单5-16展示了一个示例。

代码清单5-16　客户端代码只会在每月的双数日执行Something方法

```
public class DateTester
{
    public bool TodayIsAnEvenDayOfTheMonth
    {
        get
        {
            return DateTime.Now.Day % 2 == 0;
        }
    }
}
// . . .
class PredicatedDecoratorExample
{
    public PredicatedDecoratorExample(IComponent component)
    {
        this.component = component;
    }
```

```
        public void Run()
        {
            DateTester dateTester = new DateTester();
            if (dateTester.TodayIsAnEvenDayOfTheMonth)
            {
                component.Something();
            }
        }

        private readonly IComponent component;
    }
```

上面示例中的DateTester类是谓词修饰器类的一个依赖项。第一次重构的目标代码如代码清单5-17所示。但是，它还只是一个不完整的方案。

代码清单5-17　将依赖传入类的改进

```
    class PredicatedDecoratorExample
    {
        public PredicatedDecoratorExample(IComponent component)
        {
            this.component = component;
        }

        public void Run(DateTester dateTester)
        {
            if (dateTester.TodayIsAnEvenDayOfTheMonth)
            {
                component.Something();
            }
        }

        private readonly IComponent component;
    }
```

现在你要求给Run方法传入一个DateTester参数，但这样就破坏了客户端的公共接口设计，并要求它的客户端去实现DateTester类。此时，如果应用修饰器模式，你依然能保持现有客户端公共接口的设计，同时也能保持根据条件执行动作的能力。代码清单5-18证明了这样重构后的结果依然不够好。

代码清单5-18　谓词修饰器包含了依赖，客户端代码接口保持不变而且实现更加简洁了

```
    public class PredicatedComponent : IComponent
    {
        public PredicatedComponent(IComponent decoratedComponent, DateTester dateTester)
        {
            this.decoratedComponent = decoratedComponent;
            this.dateTester = dateTester;
        }
```

```
    public void Something()
    {
        if(dateTester.TodayIsAnEvenDayOfTheMonth)
        {
            decoratedComponent.Something();
        }
    }

    private readonly IComponent decoratedComponent;
    private readonly DateTester dateTester;
}
// . . .
class PredicatedDecoratorExample
{
    public PredicatedDecoratorExample(IComponent component)
    {
        this.component = component;
    }

    public void Run()
    {
        component.Something();
    }

    private readonly IComponent component;
}
```


上面示例中将条件分支添加到了谓词修饰器中，并没有改动客户端代码或原有的其他实现类。虽然也给谓词修饰器类引入了对DateTester类的依赖，但是你可以通过定义专门的谓词接口来更加通用地处理这种具有条件分支的场景。代码清单5-19展示的是重构后的代码。

代码清单5-19　定义一个被修饰的IPredicate接口，可以让解决方案更加通用

```
public interface IPredicate
{
    bool Test();
}
// . . .
public class PredicatedComponent : IComponent
{
    public PredicatedComponent(IComponent decoratedComponent, IPredicate predicate)
    {
        this.decoratedComponent = decoratedComponent;
        this.predicate = predicate;
    }

    public void Something()
    {
        if (predicate.Test())
        {
            decoratedComponent.Something();
```

```
            }
        }

        private readonly IComponent decoratedComponent;
        private readonly IPredicate predicate;
    }
    // . . .
    public class TodayIsAnEvenDayOfTheMonthPredicate : IPredicate
    {
        public TodayIsAnEvenDayOfTheMonthPredicate(DateTester dateTester)
        {
            this.dateTester = dateTester;
        }

        public bool Test()
        {
            return dateTester.TodayIsAnEvenDayOfTheMonth;
        }

        private readonly DateTester dateTester;
    }
```

现在由实现了`IPredicate`接口的`TodayIsAnEvenDayOfTheMonthPredicate`类来依赖`DateTester`类。这也是5.1.2节中讨论过的适配器模式的示例。

> **注意**　.NET Framework从2.0版开始就提供了一个`Predicate<T>`委托，它对谓词进行了建模，可以接受单个泛型参数作为谓词的上下文。我没有在上面的示例中选择使用`Predicate<T>`，这有两个原因：第一，并不需要任何上下文信息，因为原有的条件测试并不需要参数。然而，我依然可以使用一个`Func<bool>`委托来表示一个无需上下文的谓词，这也引出了我不选用它的第二个理由：**委托并不像接口那样通用**。通过设计`IPredicate`接口，我能够在后期用同样的方式来修饰它。也就是说，接口提供了一个可以不断扩展修饰的切入点。

5.2.3　分支修饰器

你还可以通过接受另一个接口实例并在条件判断不为真的分支下完成某些动作来对谓词修饰器做进一步的扩展，如代码清单5-20所示。

代码清单5-20　分支修饰器接受两个组件和一个谓词

```
    public class BranchedComponent : IComponent
    {
        public BranchedComponent(IComponent trueComponent, IComponent falseComponent,
    IPredicate predicate)
        {
            this.trueComponent = trueComponent;
```

```
        this.falseComponent = falseComponent;
        this.predicate = predicate;
    }

    public void Something()
    {
        if (predicate.Test())
        {
            trueComponent.Something();
        }
        else
        {
            falseComponent.Something();
        }
    }

    private readonly IComponent trueComponent;
    private readonly IComponent falseComponent;
    private readonly IPredicate predicate;
}
```


无论何时调用谓词进行判断，如果返回值为真，就调用trueComponent实例的Something方法，如果返回值不为真，就调用falseComponnet实例的Something方法。

5.2.4 延迟修饰器

延迟修饰器允许客户端提供某个接口的引用，但是直到第一次使用它时才进行实例化。通常客户端直到看到传入的Lazy<T>参数时才会觉察到延迟实例的存在，但是不应该让它们知道有延迟实例存在的细节信息。代码清单5-21展示了一个这样的示例。

代码清单5-21 客户端接受了一个Lazy<T>参数

```
public class ComponentClient
{
    public ComponentClient(Lazy<IComponent> component)
    {
        this.component = component;
    }

    public void Run()
    {
        component.Value.Something();
    }

    private readonly Lazy<IComponent> component;
}
```

上面示例中的客户端代码只有一个构造函数，且只能接受延迟实例化的IComponent接口的实例。然而，基于该接口更标准的使用方式，你还可以选择创建一个延迟修饰器，这样就可以防

止客户端知道正在处理的是Lazy<T>实例，而且也允许一些ComponentClient对象接受非延迟实例化的IComponent实例。代码清单5-22展示了延迟修饰器的实现。

代码清单5-22　LasyComponent类是IComponent接口的一个延迟实例化的实现，但是ComponentClient并不知道这些细节

```
public class LazyComponent : IComponent
{
    public LazyComponent(Lazy<IComponent> lazyComponent)
    {
        this.lazyComponent = lazyComponent;
    }

    public void Something()
    {
        lazyComponent.Value.Something();
    }

    private readonly Lazy<IComponent> lazyComponent;
}
// . . .
public class ComponentClient
{

    public ComponentClient(IComponent component)
    {
        this.component = component;
    }

    public void Run()
    {
        component.Something();
    }

    private readonly IComponent component;
}
```

5.2.5　日志记录修饰器

代码清单5-23展示的示例是代码包含大量日志语句的一种常见情况。日志代码在应用程序中无处不在，严重地拉低了有效代码率。

代码清单5-23　日志代码影响了方法意图的连贯表达

```
public class ConcreteCalculator : ICalculator
{
    public int Add(int x, int y)
    {
        Console.WriteLine("Add(x={0}, y={1})", x, y);
```

```
            var addition = x + y;

            Console.WriteLine("result={0}", addition);

            return addition;
        }
    }
```

你可以专门实现一组日志记录修饰器的程序集来解决这种应用程序中日志代码满天飞的问题，如代码清单5-24所示。

代码清单5-24 日志记录修饰器能提取出日志语句，让方法功能的实现看起来更简洁

```
public class LoggingCalculator : ICalculator
{
    public LoggingCalculator(ICalculator calculator)
    {
        this.calculator = calculator;
    }

    public int Add(int x, int y)
    {
        Console.WriteLine("Add(x={0}, y={1})", x, y);

        var result = calculator.Add(x, y);

        Console.WriteLine("result={0}", result);

        return result;
    }

    private readonly ICalculator calculator;
}
// . . .
public class ConcreteCalculator : ICalculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

`ICalculator`接口的客户端会传入多个参数，有些接口方法本身也会有返回值。因为`LoggingCalculator`类处于客户端和接口之间，因此它可以将二者直接联系起来。当然，日志记录修饰器的使用有一些局限性需要注意。第一，被修饰类中的所有私有状态一样对日志记录修饰器不可见，因此也无法将它们写入日志。第二，应用程序中的每个接口都要有对应的日志记录修饰器，这个任务工作量太过巨大。为了实现同样的目的，应该用日志记录方面来代替日志记录修饰器。面向切面编程（AOP）在第2章中有过讲解。

5

5.2.6　性能修饰器

基于.NET Framework平台开发应用的最大优势就是它能很好地支持快速应用开发（Rapid Application Development，RAD）。与诸如C++等低级编码语言相比，使用C#开发同一个可工作的应用程序需要的时间要少很多。此外，它还有诸如.NET Framework的自动内存管理，丰富的可用库，以及强大的.NET Framework本身等其他一些优势。通常情况下，C#被认为有助于提高开发效率，但开发出的应用程序运行速度相对却比较慢。相反，C++会被认为开发效率偏慢，但开发出的应用程序的运行速度却比C#快很多。

尽管基于.NET Framework开发应用程序的效率比较高，但它在运行时很可能存在性能瓶颈。那么，你又如何知道哪些代码的性能比较差呢？通过对应用程序进行性能分析，你能得知哪些代码比其他代码性能差的统计结果。先来看看代码清单5-25的示例代码。

代码清单5-25　（故意设计出的）性能比较差的代码

```
public class SlowComponent : IComponent
{
    public SlowComponent()
    {
        random = new Random((int)DateTime.Now.Ticks);
    }

    public void Something()
    {
        for(var i = 0; i<100; ++i)
        {
            Thread.Sleep(random.Next(i) * 10);
        }
    };

    private readonly Random random
}
```

上面示例中的Something方法性能很差。当然，性能的好坏都是相对的。现在这个示例里，一个性能差的方法定义是执行时间超过了一秒钟。那又如何判断一个方法是否符合这个性能差的定义呢？你可以通过对方法执行始末计时来判定，如代码清单5-26所示。

代码清单5-26　System.Disgnostics.Stopwatch类可以对方法执行时间进行计时

```
public class SlowComponent : IComponent
{
    public SlowComponent()
    {
        random = new Random((int)DateTime.Now.Ticks);
        stopwatch = new Stopwatch();
    }

    public void Something()
    {
        stopwatch.Start();
```

```
            for(var i = 0; i<100; ++i)
            {
                System.Threading.Thread.Sleep(random.Next(i) * 10);
            }
            stopwatch.Stop();
            Console.WriteLine("The method took {0} seconds to complete",
      stopwatch.ElapsedMilliseconds / 1000);
        }

        private readonly Random random;
        private readonly Stopwatch;
    }
```

这里使用的Stopwatch类包含在System.Diagnosticas程序集中，它可以用来对每个方法计时。可以看到上面示例代码中的Something方法在入口启动了秒表，在出口停止了秒表。

当然，可以把这个功能提取到一个性能修饰器中。对整个要测试性能的接口进行修饰，而且在委托被修饰的实例前，启动秒表。在被修饰实例的方法返回后，停止秒表。代码清单5-27展示了使用秒表的性能修饰器的代码。

代码清单5-27　性能修饰器的代码

```
    public class ProfilingComponent : IComponent
    {
        public ProfilingComponent(IComponent decoratedComponent)
        {
            this.decoratedComponent = decoratedComponent;
            stopwatch = new Stopwatch();
        }

        public void Something()
        {
            stopwatch.Start();
            decoratedComponent.Something();
            stopwatch.Stop();
            Console.WriteLine("The method took {0} seconds to complete",
      stopwatch.ElapsedMilliseconds / 1000);
        }

        private readonly IComponent decoratedComponent;
        private readonly Stopwatch stopwatch;
    }
```

在此基础上，还可以对ProfillingComponent再做一次重构：消除性能日志语句。第一，需要把秒表启停和计算时间间隔的代码提取并隐藏在一个接口后，这样你就可以提供多种实现，包括修饰器。这通常就是在朝着更好的职责划分目标进行重构时的第一步。代码清单5-28展示了这个中间状态。

代码清单5-28　在实现修饰器前，你必须先用接口替换具体的实现

```
    public class ProfilingComponent : IComponent
```

```
{
    public ProfilingComponent(IComponent decoratedComponent, IStopwatch stopwatch)
    {
        this.decoratedComponent = decoratedComponent;
        this.stopwatch = stopwatch;
    }

    public void Something()
    {
        stopwatch.Start();
        decoratedComponent.Something();
        var elapsedMilliseconds = stopwatch.Stop();
        Console.WriteLine("The method took {0} seconds to complete", elapsedMilliseconds /
  1000);
    }

    private readonly IComponent decoratedComponent;
    private readonly IStopwatch stopwatch;
}
```

现在，ProfillingComponent类不在直接依赖系统的System.Diagnostics.StopWatch类，你可以更改IStopWatch接口的实现。基于IStopWatch接口实现的LoggingStopwatch修饰器，能够为后续IStopWatch的实现提供日志功能，如代码清单5-29所示。

代码清单5-29　LoggingStopwatch修饰器类是IStopwatch的一个实现，它记录日志并委托其他IStopwatch实现完成真正的计时动作

```
public class LoggingStopwatch : IStopwatch
{
    public LoggingStopwatch(IStopwatch decoratedStopwatch)
    {
        this.decoratedStopwatch = decoratedStopwatch;
    }

    public void Start()
    {
        decoratedStopwatch.Start();
        Console.WriteLine("Stopwatch started...");
    }

    public long Stop()
    {
        var elapsedMilliseconds = decoratedStopwatch.Stop();
        Console.WriteLine("Stopwatch stopped after {0} seconds",
  TimeSpan.FromMilliseconds(elapsedMilliseconds).TotalSeconds);
        return elapsedMilliseconds;
    }

    private readonly IStopwatch decoratedStopwatch;
}
```

当然，你需要一个IStopwatch接口的非修饰器实现，它会完成真正的秒表功能。代码清单5-30展示的示例直接利用了.NET Framework的System.Diagnostics.Stopwatch类。

代码清单5-30 下面IStopwatch接口的实现主要使用了已有的Stopwatch类

```
public class StopwatchAdapter : IStopwatch
{
    public StopwatchAdapter(Stopwatch stopwatch)
    {
        this.stopwatch = stopwatch;
    }

    public void Start()
    {
        stopwatch.Start();
    }

    public long Stop()
    {
        stopwatch.Stop();
        var elapsedMilliseconds = stopwatch.ElapsedMilliseconds;
        stopwatch.Reset();
        return elapsedMilliseconds;
    }

    private readonly Stopwatch stopwatch;
}
```


注意，你也可以将IStopwatch的实现作为System.Diagnostics.Stopwatch类的子类以利用它已有的Start和Stop方法。然而，Stopwatch类的Start方法在秒表停止后再次调用的意图是继续前一次的计时，因此你需要在调用Stopwatch类的Stop方法以及取出ElapsedMilliseconds属性值后，立即调用它的Reset方法。这也是适配器模式的另一个示例。

5.2.7 异步修饰器

异步方法是指与客户端代码运行在不同线程上的方法。异步方式在方法执行需要很长时间的情况下很有用，因为在同步执行期间，客户端代码会被完全阻塞直到从该方法返回。比如，在使用WPF和MVVM模式的桌面应用程序里，绑定在视图上的视图模型是运行在用户界面线程上的，它包含的所有命令都是以同步方式进行处理的。在实际应用中，这就意味着长时间运行的命令会阻塞用户界面直到该命令完成它的工作。代码清单5-31展示了这种阻塞现象的代码示例。

代码清单5-31 用户界面线程上的命令会阻塞用户界面，从而导致用户界面不响应

```
public class MainWindowViewModel : INotifyPropertyChanged
{
    public MainWindowViewModel(IComponent component)
    {
        this.component = component;
```

```
            calculateCommand = new RelayCommand(Calculate);
        }

        public string Result
        {
            get
            {
                return result;
            }
            private set
            {
                if (result != value)
                {
                    result = value;
                    PropertyChanged(this, new PropertyChangedEventArgs("Result"));
                }
            }
        }

        public ICommand CalculateCommand
        {
            get
            {
                return calculateCommand;
            }
        }

        public event PropertyChangedEventHandler PropertyChanged = delegate { };

        private void Calculate(object parameter)
        {
            Result = "Processing...";
            component.Process();
            Result = "Finished!";
        }

        private string result;
        private IComponent component;
        private RelayCommand calculateCommand;
    }
```

通过创建一个异步修饰器，你可以指示被调用的方法在一个单独的线程中执行。这可以通过将具体工作委托给一个Task类的实例来实现，它也会变成你的异步修饰器的依赖，如代码清单5-32所示。

代码清单5-32　一个为WPF使用Dispatcher类的异步修饰器

```
    public class AsyncComponent : IComponent
    {
        public AsyncComponent(IComponent decoratedComponent)
        {
            this.decoratedComponent = decoratedComponent;
```

```
        }

        public void Process()
        {
            Task.Run((Action)decoratedComponent.Process);
        }

        private readonly IComponent decoratedComponent;
    }
```

示例中的AsyncComponent类有一个问题：它隐式依赖Task类，这就意味着很难测试它。使用静态依赖的代码很难进行单元测试，因此你最好选个塔吊替代这个天钩。

异步修饰器的局限性

并不是所有方法都可以使用修饰器模式提供给客户端并不可见的异步版本。实际上，只有那些即发即弃（fire-and-forget）的方法才可以应用异步修饰器。

一个即发即弃的方法并没有返回值，客户端代码无需知道这个方法何时返回调用。如果一个方法实现为异步修饰器，客户端代码就无法知道该方法什么时候才真正完成，因为对它的调用会立即返回，实际上真正要执行的工作很可能依然在进行中。

请求-响应（request-response）方法通常用来获取数据，也经常会被实现为异步方法，因为这些方法通常会花点时间且阻塞UI线程。客户端需要知道该方法是异步的，这样它们就可以显式编写回调以便在该异步方法完成时得到通知。因此，请求-响应方法并不适合使用异步修饰器来实现。

5.2.8 修饰属性和事件

到目前为止，你学到的都是如何修饰接口的方法，那么属性和事件能被修饰吗？答案是肯定的。为了修饰它们，你需要显式定义它们而不是使用自动属性和自动事件。

代码清单5-33展示了手动创建的属性，它的存取器直接委托被修饰实例的存取器，而不是使用一个后备字段。

代码清单5-33 属性也可以像方法一样使用修饰器模式

```
    public class ComponentDecorator : IComponent
    {
        public ComponentDecorator(IComponent decoratedComponent)
        {
            this.decoratedComponent = decoratedComponent;
        }

        public string Property
        {
            get
            {
                // We can do some mutation here after retrieving the value
                return decoratedComponent.Property;
```

```
        }
        set
        {
            // And/or here, before we set the value
            decoratedComponent.Property = value;
        }
    }

    private readonly IComponent decoratedComponent;
}
```

代码清单5-34展示了一个手动创建的事件，它的添加器和移除器直接委托被修饰实例的事件的添加和移除方式，而不是使用一个后备字段。

代码清单5-34　事件也可以像方法一样使用修饰器模式

```
public class ComponentDecorator : IComponent
{
    public ComponentDecorator(IComponent decoratedComponent)
    {
        this.decoratedComponent = decoratedComponent;
    }

    public event EventHandler Event
    {
        add
        {
            // We can do something here, when the event handler is registered
            decoratedComponent.Event += value;
        }
        remove
        {
            // And/or here, when the event handler is deregistered
            decoratedComponent.Event -= value;
        }
    }

    private readonly IComponent decoratedComponent;
}
```

5.3　用策略模式替代 switch 语句

为了理解应用策略模式的最佳时机，你可以看看条件分支的场景。任何使用switch语句的场合，你都可以通过策略模式将复杂性委托给所依赖的接口以简化客户端代码。代码清单5-35展示了一个可以使用策略模式替换的switch语句代码示例。

代码清单5-35　这个方法使用了switch语句，但是策略模式能提供更好的适应变更的能力

```
public class OnlineCart
```

```
{
    public void CheckOut(PaymentType paymentType)
    {
        switch(paymentType)
        {
            case PaymentType.CreditCard:
                ProcessCreditCardPayment();
                break;
            case PaymentType.Paypal:
                ProcessPaypalPayment();
                break;
            case PaymentType.GoogleCheckout:
                ProcessGooglePayment();
                break;
            case PaymentType.AmazonPayments:
                ProcessAmazonPayment();
                break;
        }
    }

    private void ProcessCreditCardPayment()
    {
        Console.WriteLine("Credit card payment chosen");
    }

    private void ProcessPaypalPayment()
    {
        Console.WriteLine("Paypal payment chosen");
    }

    private void ProcessGooglePayment()
    {
        Console.WriteLine("Google payment chosen");
    }

    private void ProcessAmazonPayment()
    {
        Console.WriteLine("Amazon payment chosen");
    }
}
```


示例中，对应switch语句下的每个case分支，类的行为都有变化。这种方式会引入代码维护问题，因为增加任何新的case分支都需要更改这个类。相反，如果你用同一接口的一组实现来替代所有的case分支，那么后续还可以增加新的实现来封装新的功能，而且客户端代码也无需改变。代码清单5-36展示了这样的重构。

代码清单5-36 替换switch语句后，客户端代码看起来适应变更的能力高了很多

```
public class OnlineCart
{
    public OnlineCart()
    {
```

```
            paymentStrategies = new Dictionary<PaymentType, IPaymentStrategy>();
            paymentStrategies.Add(PaymentType.CreditCard, new PaypalPaymentStrategy());
            paymentStrategies.Add(PaymentType.GoogleCheckout, new
      GoogleCheckoutPaymentStrategy());
            paymentStrategies.Add(PaymentType.AmazonPayments, new
      AmazonPaymentsPaymentStrategy());
            paymentStrategies.Add(PaymentType.Paypal, new PaypalPaymentStrategy());
        }

        public void CheckOut(PaymentType paymentType)
        {
            paymentStrategies[paymentType].ProcessPayment();
        }

        private IDictionary<PaymentType, IPaymentStrategy> paymentStrategies;
    }
```

按照面向对象编程的传统，示例代码将不同的支付类型具体化为不同的类，它们都实现了IPaymentStrategy接口。这个示例中，OnlineCart类带有一个私有的字典字段，它可以将PaymentType枚举的每个值映射到一个对应的IPaymentStrategy接口的实例上。这个字典大大地简化了Checkout方法的复杂度。不仅switch语句被移除了，各种类的支付处理过程也随之不复存在。OnlineCart类不需要知道如何处理支付，大量不同的处理方式会给该类引入太多不必要的依赖。现在的OnlineCart类只是选择恰当的支付策略并委托它来完成实际处理过程。

在添加新的支付策略实现时，该示例依然存在一个维护负担。比如，在添加实现以支持WePay时，你就需要更改构造函数以映射新的WePayPaymentStrategy类给对应的WePay枚举值。

5.4　总结

单一职责原则对代码自适应能力有着至关重要的正面影响。与不应用该模式的同样功能的代码相比，符合单一职责原则的代码会由更多的小规模但目标更明确的类组成。单个巨型类或相互依赖的一组类只会导致职责混淆，而单一职责原则能带来有序和清晰的良好效果。

单一职责原则主要是通过接口抽象以及在运行时将无关功能的责任委托给相应接口完成来达成目标的。一些设计模式（特别是适配器模式和修饰器模式）非常适合支持单一职责类的实现。适配器模式能让你的绝大多数代码都引用你能完全控制的第一方组件，尽管实际上是在利用第三方库。当一个类的某些功能需要被移除但这些功能又和该类意图紧密联系时，就可以应用修饰器模式。

本章并没有讲解所有这些类运行时的协作方式。本章采用的方式是将接口实例传入构造函数的方式，后面第9章将会讲解能实现同样目标的多种不同的方式。

第 6 章

开放与封闭原则

完成本章学习之后，你将学到以下技能。

- 理解开放与封闭原则的不同解释。
- 只对SOLID代码做追加修改。
- 比较和对比不同的类扩展机制。
- 将防止变异作为扩展点的指导原则。

开放与封闭原则自相矛盾的本质会引起很多困惑。顾名思义，该原则要求代码既是宽容开放的，同时又是严格有限制的。它的几种变体只会出现在云相关的场景中。

总是选择并使用一种定义对我来说是不可以接受的。所以，本章中我比较了所有定义和它们相应的影响，以便挖掘出这个原则的本质。清楚该原则的本质有助于你编写出具有更强自适应能力的代码。

6.1 开放与封闭原则介绍

有两种不同的开放与封闭原则定义必须要介绍到，它们是20世纪80年代的最原始定义和后期一个更现代的定义。基于更多的上下文和原则范围定义，后者尝试了对前者进行更加详尽的阐述。

6.1.1 Meyer 的定义

Bertrand Mayer在他的著作《面向对象软件构造》中首次定义了开放与封闭原则（Open/Closed Principle，OCP），具体定义如下所示。

> 软件实体应该允许扩展，但禁止修改。

Mayer的定义是被引用最多的该原则的定义，但是Martin也给出了另外一个定义。

6.1.2 Martin 的定义

Robert C. Martin定义的开放与封闭原则在过去的十几年里有很多种不同的写法。这里选择的是一个非常详细的版本，摘自《敏捷软件开发：原则、模式与实践》，以便和简短的原始版本进

行对比。

> “对于扩展是开放的。”这意味着模块的行为是可以扩展的。当应用程序的需求改变时，我们可以对其模块进行扩展，使其具有满足那些需求变更的新行为。换句话说，我们可以改变模块的功能。
>
> “对于修改是封闭的。”对模块行为进行扩展时，不必改动该模块的源代码或二进制代码。模块的二进制可执行版本，无论是可链接的库、DLL或Java的.jar文件，都无需改动。

Martin详细解释了Meryer定义中的开放与封闭原则的两个关键词。对于扩展是开放的，Martin的解释是，开发人员必须能够响应需求变更并支持新的特性。必须要做到这一点，尽管模块对修改是封闭的。开发人员必须在不改动已有模块源代码或程序集的前提下支持新的功能。

在继续讲解如何做到开放与封闭原则要求的两点前需要指出，其中经常被引用的“对于修改是封闭的”一句也有两个例外：修复缺陷所做的改动以及客户端无法感知到的改动。

6.1.3　缺陷修复

缺陷在软件中很常见，是不可能完全消除的。但当它们出现时，你需要修复问题代码。当然，这会牵扯到修改现有的类，除非你愿意为修复问题将现有类的实现在新版本中再复制一份。缺陷修复的方式听起来似乎有些令人费解，它明显倾向于实用主义而不是坚持纯正的开放与封闭原则。

下面列出修复缺陷流程的两个步骤。

(1) 针对缺陷编写失败的单元测试和/或集成测试。前提条件是要有稳定的让代码失败的问题复现步骤。根据前面章节提到的单元测试的布置、动作和断言模式定义的语法，你需要能先布置好测试目标系统以便让它处在能触发缺陷的状态，然后执行指定的包含缺陷的动作，最后对期望的行为进行断言。这种单元测试开始时总是失败的。这就说明了所有问题实际上是由缺乏测试引起的。如果有测试用于捕获这个缺陷，那么该测试会是失败的，当然，如果使用了持续集成系统，它也会报错。

(2) 修改后的源代码才可以通过单元测试。在这种特定情况下，违背开放与封闭原则的缺陷修复也是很有必要的，因为如果没有它，你将无法修改任何现有代码。通过修改测试目标系统的实现能让失败的单元测试由红色失败状态变为绿色成功状态。当你能确认没有产生任何副作用，也就是不会导致其他任何测试失败时，这个缺陷就已经被成功修复了。

6.1.4　客户端感知

一个更加违背“对于修改是封闭的”规则的例外情况是：允许对现有代码做任意改动，只要它不会引起对任意客户端代码的改动。它着重强调软件模块在所有粒度级别上如何耦合关联，包括类之间、程序集之间以及子系统之间。

如果一个类的改动会引起另外一个类的改动，那么这两个类就是紧密耦合的。相反，如果一

个类的改动是独立的，并不会引起其他类的改动，那么这些类就是松散耦合的。任何情况下，松散耦合都比紧密耦合要好。如果你对现有代码的修改不会影响客户端代码，那么维护这样松散耦合的代码对开放与封闭原则的影响是有限的。

6.2　扩展点

前面已经对开放与封闭原则的“对于修改是封闭的”规则做了澄清，下面开始讲解“对于扩展是开放的”这个规则。应用了开放与封闭原则的类应该通过定义扩展点来对扩展保持开放，这些现有代码上的扩展点可以在将来引入新的功能以提供新的行为。

本节会详细讲解一些不同类的扩展点及其优缺点。本章会在前一章使用的TradeProcessor类的示例上继续讲解，讲解的重点是客户端代码和该类的交互。

6.2.1　没有扩展点的代码

首先，没有扩展点的代码是什么样子的呢？图6-1展示了没有扩展点的类需要添加新功能时的状况。

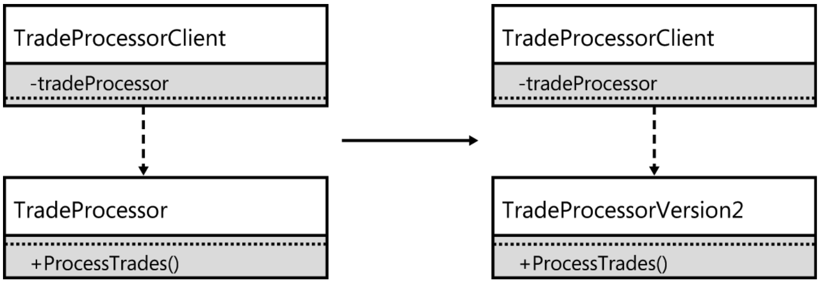


图6-1　如果没有扩展点，则会强制客户端进行更改

TradeProcessorClient类直接依赖TradeProcessor类。当你接到一个需要改动TradeProcessor类的新需求时，为了不改变原有类型，创建了一个新版本（TradeProcessorVersion2）来包含新需求提出的新功能。客户端直接依赖TradeProcessor类并且该类没有提供任何扩展点，因此你需要将新功能安排在新的类中。这种改动的副作用就是，必须改动TradeProcessorClient类，这样才能依赖新的TradeProcessorVersion2类。

如果对现有代码的改动不会影响客户端，你也许就不需要再创建一个全新的TradeProcessor类了。如果要改变的是ProcessTrades方法的签名，那这就不是简单的对类实现的改动，而是对接口的改动了。因为客户端总是与服务的接口紧密耦合的，所以任何接口上的改动都会引起客户端代码的改动。

6.2.2　虚方法

TradeProcessor类的另外一种实现包含了一个扩展点：ProcessTrades是个虚方法。图6-2展示了使用虚方法后的三个类的关系。

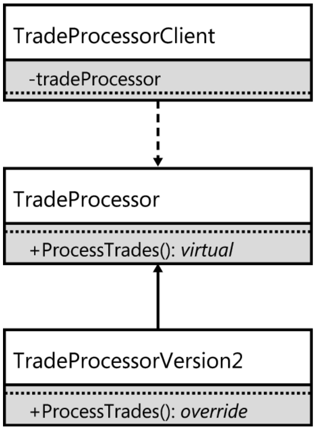


图6-2　客户端依赖TradeProcessor类，该类可以通过继承进行扩展

任何带有一个虚方法成员的类都是对扩展开放的。这种扩展是通过实现继承做到的。当TradeProcessor类的新特性需求到来时，你可以修改其子类的ProcessTrades方法而无需改变原有的TradeProcessor类源代码。

此时的TradeProcessorClient不需要做改动，因为你可以使用多态向客户端提供新版本的TradeProcessorVersion2类的实例并使其调用该实例的ProcessTrades方法。

然而，你能重新实现的范围是有一定限制的。在新的子类中，你依然能够访问基类，因此可以直接调用TradeProcessor类的ProcessTrades方法，但是无法改动该方法内的任何代码。你要么在子类方法里调用基类同名方法并且在其前或后实现新的特性，要么你完全重新实现子类的方法。虚方法并没有中间状态。记住，子类只能访问基类的受保护和公共成员。如果TradeProcessor类带有很多你无权访问的私有成员，你就需要修改该基类的实现了，当然，这样就会违背开放与封闭原则。

6.2.3　抽象方法

另外一种使用实现继承的更加灵活的扩展点是抽象方法。在这种情况下，TradeProcessor类是一个定义了公共ProcessTrades方法的抽象类，该方法会委托三个抽象的保护方法来完成交易处理算法的工作。客户端对这三个保护方法并不知情，因为它们都是没有具体实现的抽象方法。图6-3展示了相关类的关系。

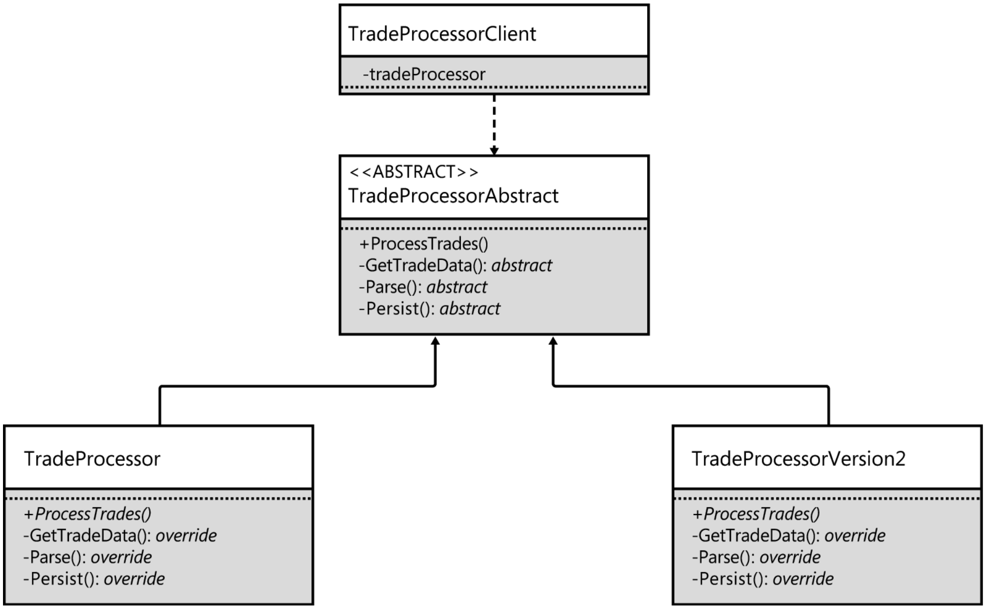


图6-3　抽象方法为将来的子类提供了扩展点

示例中提供了两个版本的交易处理器。它们都从抽象基类中直接继承了`ProcessTrades`方法，也都为三个抽象方法提供了各自的实现。客户端依赖抽象基类，因此提供任何一个具体子类（或者用来支持新需求的新子类）给客户端都不会违背开放与封闭原则。

这也是模板方法模式（Template Method pattern）的一个示例。该模式只对算法框架建模，而算法的每个具体步骤还可以自定义，因为它们都是委托抽象方法来完成具体动作的。实际上，基类委托子类来处理每个具体的步骤。

6.2.4　接口继承

本章讨论的最后一个扩展点是实现继承外的另外一种选项：接口继承。这里，客户端委托接口取代了客户端对类的依赖。图6-4展示了客户端对接口的依赖以及该接口的两个实现。

接口继承要比实现继承好很多。基于实现继承，所有子类（现有的和将来的）都是基类的客户端。这会影响后续的修改，因为子类也都是依赖基类实现的。因此，所有对实现的改动都会是客户端可能察觉到的改动。因此相对于继承，通常会建议优先选择组合，如果必须要使用继承，也要尽量使用只有少量分层的浅继承层次结构。给继承图顶部节点添加新成员的改动会影响到该层次结构下的所有成员。

接口也是一个比较好的扩展点，因为可以根据不同的上下文给接口修饰丰富的功能。接口要比类灵活得多。虽然这并不代表类继承的虚方法和抽象方法提供的扩展点没有一点用处，但是它们的确无法提供与接口一样强大的自适应能力。

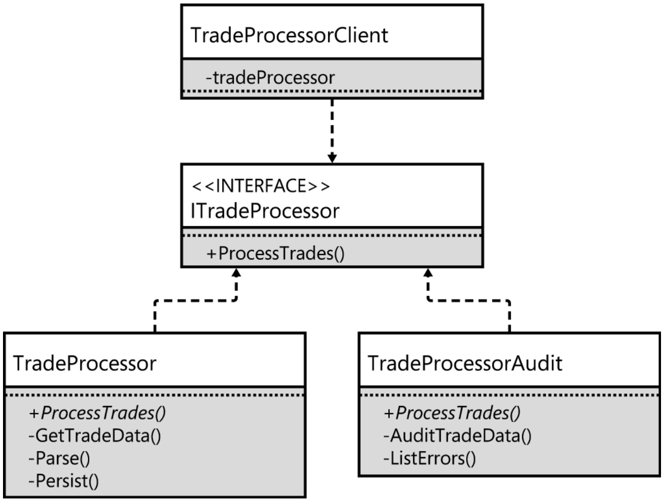


图6-4　客户端依赖接口而不是类

6.2.5　“为继承设计或禁止继承”

Joshua Bloch在他的著作《Effective Java 中文版（第2版）》中对继承的描述如下所示。

> 为继承设计和撰写文档，或者禁止使用继承。

如果你选择使用实现继承作为扩展点，就必须恰当地设计该类并为此编写清楚的文档，以便让后续要扩展该类的编程人员清楚原始的设计。类的继承会比较复杂，因为新的子类会以一种不可预期的方式破坏现有代码。

切记，任何没有标记`sealed`关键字的类都提供了继承能力。类并不是必须要有虚方法或者抽象方法才能够派生子类。使用new关键字可以隐藏被继承的成员，但是这种方式会影响多态能力的使用，显然违背了我们的期望。

通过密封类可以清楚地告诉其他可能使用该类的编程人员：该类并不支持继承。这样他们就会重新寻找替代方案。

6.3　防止变异

现在你已经有了好几个能实现开放与封闭原则的工具了。你清楚何种场合下可以修改现有代码，你也知道需要在你的代码中实现扩展点以便支持将来的需求变更。你还知道可以使用接口作为扩展点来让你的代码变得真正具有自适应能力，甚至可以应对将来的考验。

我们还没有讲解到应用开放与封闭原则的时机和场合。极端地想，你应该到处都留着扩展点吗？这样做会让你的代码变得无限灵活，还是应用效果会逐渐递减？

下面是另外一个跟开放与封闭原则相关的重要准则：防止变异（protected variation）。这是Alistair Cockburn提出的一个术语。

> 识别可预见的变化点并围绕它们创建一个稳定的接口。

该原则本身叫作防止变异，而上面的定义引用了术语可预见的变化（predicted variation），这多少会让人产生一些疑惑。尽管如此，我脑海里中仍然认为“可预见的变化”会更加合适一些。接下来会详细讲解该定义的两个方面。

6.3.1 可预见的变化

单个类的需求应该直接与客户端的一个业务需求关联起来。如果忽略了这种关联，这个类就很有可能不是在为客户端要求的业务目标服务的。冲刺过程中，开发人员从冲刺积压工作上去除用户故事，然后与产品负责人沟通故事相关的事情。此时，开发人员就可以提出有关将来的、潜在的相关需求。这样构造得到的可预见变化是可以直接转化为扩展点的。

6.3.2 一个稳定的接口

即使你只委托接口，客户端仍然依赖这些接口。如果接口发生变化，客户端也必须做相应的改动。依赖接口的最大优势是接口变化的可能性要比实现小很多。如果你按照阶梯模式将接口和实现分别组织在不同的程序集中，那么二者能够独立变化而不相互影响，而且实现的变动也不会影响到接口的客户端。

显然，用于表达扩展点的所有接口应该是稳定的，这一点非常重要。接口改变的可能性和频率应该很低，否则你会需要通知所有客户端使用新版本的接口了。

6.3.3 足够的自适应能力

只在合适的位置上包含恰当数目的扩展点的代码也被称为代码的“宜居带”，它能适应代码后续的变更需求，同时又不会增加复杂度和过度设计方案。对于任何具体问题而言，代码的自适应能力要么不够，要么过度，要么正好。

大多数编程“新手”通常会以过程化方式编码，即使使用的是诸如C#等支持面向对象的语言。他们会习惯将类看作方法的聚集地，而不管这些方法在职责上是否真的应该在一起。他们几乎不做架构就直接编码，代码中也没有多少扩展点（即使有，也都没用对地方）。任何需求变更都会直接导致对现有类代码的修改。第5章开篇给出的原始`TradeProcessor`类就是这些“新手”们常见的产出。若要使用那个原始类，你就必须要“精通”那段代码相关的一切信息。

尽管如此，有时采取这种简单的实现也不算错。如果你在评估了一个诸如`TradeProcessor`之类的小工具应用程序后认为它几乎不会再发生任何改变，那么原始版本的过程化实现代码就足够了。在原始版本上针对清晰度重构后的新版本很可能也可以满足需求。如果你根本不打算对此进行扩展，那么针对抽象的重构所付出的努力将是不值得的。而且重构后的代码实现会隐藏在接

口后并且布局在多个文件和程序集中，从而导致代码在一定程度上不那么直观可读。

另一种极端情况是，刚刚喜欢上抽象的编程“老手”会尝试到处应用手中这个强大的工具。他们编写的代码总是有很多扩展点，而绝大多数根本就不会被用到。为了提供这么多的扩展点，他们总是在编码过程中花费大量的精力组织代码和委托责任给接口。

如果把上面的“新手”和“老手”的编码风格结合一下，可能编写出更加和谐的中间代码，这些代码中有足够的扩展点，但这些扩展点只针对那些需求不清楚、不稳定或难以实现的代码区域。然而，要有丰富的经验积累才能做到这一点，要经历知之甚少的“新手”和以为无所不知的“老手”阶段后，才能逐渐精通防止变异的精髓，并进阶到真正高手的层次。

6.4　总结

开放与封闭是一个面向类和接口设计的整体原则，它指导开发人员如何才能编写出很好地自适应变更的代码。每个冲刺都需要拥抱那些不期而至的新需求。然而，承认并接受变更只是解决问题的第一步。如果你的代码直到出现变更时才发现它并没有为变更做好准备，此时为了适应变更要做的工作会更困难、更费时、更易出错，代价也更高。

通过确保代码对扩展开放以及对修改封闭，你有效地阻止了后期变化对现有类和程序集的修改，因为后面的编码人员只能在你预留的扩展点上挂靠新创建的类。可用的扩展点主要有两种：实现继承和接口继承。虚方法和抽象方法允许你创建子类来定制基类的方法。如果你选择将类委托给接口，那就可以应用很多优秀的模式来更加灵活地创建和使用扩展点。

尽管如此，只知道要在代码中预留扩展点是不够的，你还需要知道应用它们的恰当时机。防止变异的概念建议你先识别出很可能发生变更的需求或者实现起来特别麻烦的代码部分，然后将它们隐藏在扩展点之后。代码可以很死板，几乎无法扩展和细化；代码也可以很流畅，带有足够的准备应对新需求的大量扩展点。两种选择并没有对错，只是要根据具体的场景和上下文进行应用。

第 7 章

Liskov替换原则

完成本章学习之后，你将学到以下技能。

- 理解Liskov替换原则的重要性。
- 避免违背Liskov替换原则的规则。
- 进一步巩固你的单一职责原则和开放与封闭原则的习惯。
- 创建遵守基类契约的派生类。
- 使用代码契约来实现前置条件、后置条件以及数据不变式。
- 编写正确引发异常的代码。
- 理解协变、逆变和不变性并知道它们的应用场合。

7.1 Liskov 替换原则介绍

Liskov替换原则（Liskov Substitution Principle，LSP）是一组用于创建继承层次结构的指导原则。按照Liskov替换原则创建的继承层次结构中，客户端代码能够放心地使用它的任意类或子类而不担心影响所期望的行为。

如果不遵守Liskov替换原则的规则，对一个类层次结构的扩展（也就是说，增加一个新的子类）很可能迫使所有使用基类或接口的客户端代码也要做相应的改动。相反，如果严格遵守Liskov替换原则的规则，客户端将无法看到对类层次结构所做的任何改动。只要接口保持不变，就应该没有理由改动任何已有的客户端代码。因此，Liskov替换原则也辅助增强了开放与封闭原则和单一职责原则的应用效果。

7.1.1 正式定义

Liskov替换原则的正式定义是由杰出的计算机科学家Barbara Liskov给出的。因为该定义非常简短，所以这里需要做进一步的详解。下面是Liskov替换原则的正式定义。

> 如果S是T的子类型，那么所有T类型的对象都可以在不破坏程序的情况下被S类型的对象替换。

定义中有三个与Liskov替换原则相关的代码要素。

- 基类型：客户引用的类型（T）。子类型可以重写（或部分定制）客户所调用的基类的任意方法。
- 子类型：继承自基类型（T）的一组类（S）中的任意一个。客户端不应该，也不需要知道它们在实际调用哪个具体的子类型。无论使用的是哪个子类型实例，客户端代码所表现的行为都是一样的。
- 上下文：客户端和子类型交互的方式。如果客户端不和子类型交互，就谈不上是否违背或遵守了Liskov替换原则。

7.1.2 Liskov 替换原则的规则

要应用Liskov替换原则就必须遵守几个规则。这些规则可以划分为两类：契约规则（与类的期望有关）和变体规则（与代码中能被替换的类型有关）。

1. 契约规则

这些规则与子类型的契约及其相应的约束相关。

- 子类型不能加强前置条件。
- 子类型不能削弱后置条件。
- 子类型必须保持超类型中的数据不变式（不变式是一个必须保持为真的条件）。

为了理解这些契约规则，你必须先理解契约的概念，然后再弄清楚在创建子类型时确保遵守这些规则需要做的事情。稍后的7.2节将会深入讨论这两方面。

2. 变体规则

这些规则与方法的参数及返回类型相关。

- 子类型的方法参数类型必须是支持逆变的。
- 子类型的返回类型必须是支持协变的。
- 子类型不能引发不属于已有异常层次结构中的新异常。

支持Microsoft .NET Framework的公共语言运行时的语言的类型变体的概念只局限于泛型和委托。尽管如此，这些场景下的变体依然是值得好好探讨的，这是你必不可少的知识武器，这样才可以为类型的变体编写出符合Liskov替换原则的代码。稍后的7.3节会深入讨论这个主题。

7.2 契约

我们经常会说，开发人员应该面向接口编程或面向契约编程。然而，除了表面上的方法签名，接口所表达的只是一个不够严谨的契约概念。如图7-1所示，只从方法的签名很难看到很多与实际需求以及方法实现保证相关的信息。在诸如C#之类的强类型化编程语言中，至少都有给参数传递正确类型的概念，但是接口距离真正的契约还很远。

所有方法至少有一个可选的返回类型、一个名称和一个可选的正式参数的列表。每个参数都由一个具体类型和名称组成。在调用图7-1中展示的方法时，你知道（只从方法签名上看）需要传入三个参数，一个类型是`float`，一个类型是`Size<float>`，另外一个类型是`RegionInfo`。你

也知道可以将类型为`decimal`的返回值保存到一个变量中或在调用结束后操作该返回值。

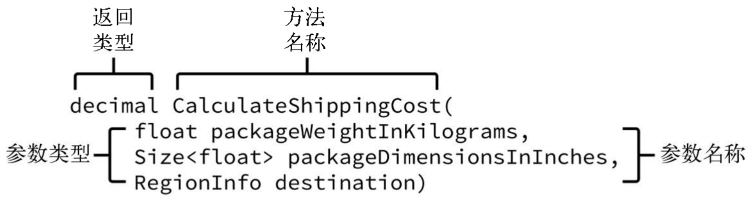


图7-1　从方法签名只能看到与实现期望相关的信息

> **注意**　图7-1中使用`decimal`类型来表示货币值是不可取的，而应该使用`Money`[①]值类型。尽管我已经努力确保本书中尽可能多的示例都不是与现实世界的上下文无关的自造概念，但仍有一些地方为了简洁而做出了妥协。

作为方法编写者，你能控制方法和参数的名称。你要特别用心地确保方法名称能反映出它的真实目的，同时参数名称要尽可能是描述性的。`CalculateShippingCost`函数的名称使用了动名词组的形式。这里的动词（由方法执行的动作）是`Calculate`，名词（动词的对象）是`ShippingCost`。在某种意义上，这个名词就是返回值的名称。参数也选择了描述性的名称：`packageDimensionsInInches`和`packageWeightInKilorams`都是自解释的参数名称，特别是在该方法的上下文中。它们构成了方法文档化的开端。

> **提示**　想了解更多有关好的变量和方法命名，以及其他最佳实践，Steve McConnell的*Code Complete*[②]一书是一个很好的选择。

然而，方法签名并没有包含方法的契约信息。比如，`packageWeightInKilograms`参数是`float`类型的。这就暗示客户端任何`float`值都是有效的，包括负数值。但是因此参数表达的是重量，所以负数值应该是无效的。方法的契约应当强制要求重量值必须大于零。为了保证做到这一点，方法必须要实现一个前置条件。

> **提示**　尽管本章中概要地讲解到的契约能够在运行时阻止很多对方法的无效调用，但良好的方法和参数名称仍然不可或缺。比如，如果`CalculateShippingCost`方法的正式参数并没有说明它们的计量单位是英寸或者千克，客户端很可能会在调用该方法时传入以厘米和磅为计量单位的值。

① http://moneytype.codeplex.com/。
② http://www.stevemcconnell.com/cc.htm。

7.2.1　前置条件

前置条件（precondition）是一个能保障方法稳定无错运行的先决条件。所有方法在被调用前都要求某些前置条件为真。默认情况下，接口并没有任何对方法具体实现的保障。代码清单7-1给你展示了如何在方法入口添加防卫子句来实现前置条件。

代码清单7-1　引发异常是一种强制履行契约的高效方式

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f) throw new Exception();

    return decimal.MinusOne;
}
```

方法入口处的`if`语句是一种强制设置前置条件的方式，比如重量必须大于零千克的需求。如果条件`packageWeightInKilograms <= 0f`为真，方法会引发一个异常并立即结束运行。这种方式肯定可以阻止方法在有参数无效的情况下被执行。通过使用一个更具描述性的异常，你能提供给调用者更多上下文信息，如代码清单7-2所示。

代码清单7-2　尽可能提供详尽的前置条件校验失败原因是很重要的

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    return decimal.MinusOne;
}
```

新的异常有了很多改进，它的名称解释了自己的目的：参数超出了有效范围，而且客户端能知道是哪个参数出错以及相应的问题描述。

通过像这样将更多的防卫子句链接到一起，你可以添加更多条件，这些条件是为了调用方法而不引发异常所必须满足的条件。代码清单7-3中显示的更改也包含包尺寸超出范围时引发的异常。

代码清单7-3　增加足够多的必要前置条件可以防止在参数无效的情况下方法被调用

```
public decimal CalculateShippingCost(
    float packageWeightInKilograms,
```

```
    Size<float> packageDimensionsInInches,
    RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

    return decimal.MinusOne;
}
```

如果有了这些前置条件，客户端代码就必须在调用方法前确保它们要传递的参数值要处于有效范围内。当然，所有在前置条件中检查的状态必须是公开可访问的。如果客户端无法验证由于未通过前置条件检查导致将要调用的方法引发异常，客户端就无法确保接下来的调用一定会成功。因此，私有状态不应该是前置条件检查的目标，只有方法参数和类的公共属性才应该有前置条件。

7.2.2 后置条件

后置条件（postcondition）会在方法退出时检测一个对象是否处于一个无效的状态。只要方法内改动了状态，就有可能因为方法逻辑错误导致状态无效。

与实现前置条件一样，可以使用防卫子句来实现后置条件。然而，后置条件并不是布置在方法的入口处，而是必须布置在所有的状态编辑动作之后的方法尾部，如代码清单7-4所示。

7

代码清单7-4 方法尾部的临界子句是一个后置条件，它能确保返回值处于有效范围内

```
public virtual decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
{
    if (packageWeightInKilograms <= 0f)
        throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
  must be positive and non-zero");

    if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
        throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

    // shipping cost calculation

    var shippingCost = decimal.One;

    if(shippingCost <= decimal.Zero)
        throw new ArgumentOutOfRangeException("return", "The return value is out of
  range");

    return shippingCost;
}
```

通过预先定义的有效范围检查状态值（如果值不在指定范围就引发异常），你能强制方法符合一个后置条件。上面示例中的后置条件与对象的状态并不相关，而是与方法返回值相关。像方法参数要经过前置条件检查一样，方法的返回值也需要经过后置条件的校验。如果方法中任意地方将返回值设置为零或者负数值，这个后置条件会检测到它并在方法尾部中止执行。通过这种方式，该方法的客户端永远无法在意外地收到无效返回值时还能认为它依然有效。注意，该方法的签名无法保证返回值必须大于零，要达到这个目的，必须通过客户端履行方法的契约来保证。

7.2.3　数据不变式

第三种类型的契约是数据不变式。*数据不变式*（data invariant）是在一个对象生命周期内始终都保持为真的一个谓词；该谓词条件在从对象构造后一直到超出其作用范围前这段时间都为真。数据不变式都是与期望的对象内部状态有关。`ShippingStrategy`调用的一个数据不变式的一个示例是：提供的比例税率为正值且不为零。如果如代码清单7-5中所示，在构造函数中设置比例税率，那么只需要在构造函数入口处增加一个防卫子句就可以防止将其设置为无效值。

代码清单7-5　给构造函数增加前置条件能够保证相应的数据不变式

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        if (flatRate <= decimal.Zero)
            throw new ArgumentOutOfRangeException("flatRate", "Flat rate must be positive
  and non-zero");

        this.flatRate = flatRate;
    }

    protected decimal flatRate;
}
```

因为`flatRate`是一个受保护的成员变量，所以客户端只能通过构造函数来设置它。如果传入构造函数的值是有效的，这就保证了该`ShippingStrategy`类实例的对象在生命周期内`flatRate`值都是有效的，因为客户端没有其他方式可以修改它。

然而，如果把`flatRate`定义为公共且可设置的属性，为了保护这个数据不变式，就必须把防卫子句布置到属性设置器内。代码清单7-6展示了重构为公共属性的`FlatRate`和相应的防卫子句。

代码清单7-6　当数据不变式是一个公共属性时，防卫子句就应该布置在它的设置器中

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        FlatRate = flatRate;
```

```
        }

        public decimal FlatRate
        {
            get
            {
                return flatRate;
            }
            set
            {
                if (value <= decimal.Zero)
                    throw new ArgumentOutOfRangeException("value", "Flat rate must be positive
      and non-zero");

                flatRate = value;
            }
        }
    }
```

现在客户端能够改变FlatRate属性的值了，但是由于属性设置器内的if语句和异常的保护，这个数据不变式是无法被破坏的。

封装与契约

虽然该示例中的契约实现是有意义的，但这里为每个值选择的类型都不太恰当。为了确保包裹重量参数的数值必须大于零而引入的前置条件契约已经和变量的类型本质有了关联：重量应该大于零千克。这其实已经意味着应该为重量封装一个专有的类型了。如果，也很有可能，另外一个类或者方法也需要一个重量值，如果没有定义类，你就不得不把所有的前置条件语句再搬到新的实现代码中。这样做效率低，难维护而且容易出错。这时为它专门定义一个包括该前置条件的Weight类型就很有意义了，这样所有使用Weight类型的客户端都必须传入一个大于零的值。实际上，Weight类型的数据不变式要比CalculateShippingCost方法中的前置条件更好。

类似地，使用decimal类型对比例税率建模也不是很好。相反，也应该为比例税率创建一个专有的类型，该类型包含了一个确保数值必须大于零的数据不变式。

7

7.2.4　Liskov契约规则

前面讨论的方法契约仅仅是Liskov替换原则的一小部分。Liskov替换原则设置了一组类型必须继承契约的规则。作为提醒，下面再次列出了Liskov替换原则的正式定义。

> 如果S是T的子类型，那么所有T类型的对象都可以在不破坏程序的情况下被S类型的对象替换。

通过这个定义可以导出以下与契约相关的指导原则（前面一节也介绍过）。

- 子类型不能加强前置条件。
- 子类型不能削弱后置条件。
- 子类型必须保持超类型中的数据不变式。

如果你在基于现有类创建子类时遵守了所有这些规则，那么替换性将会在你处理契约时得到保留。

任何时候创建子类，都能带有所有组成父类的方法、属性和字段。当然，也包括方法和属性设置器内的所有契约。所有前置条件、后置条件和数据不变式都被期望按照父类中的相同方式保留。在适当的时候，子类被允许重写父类的方法实现，此时才有机会修改其中的契约。Liskov替换原则明确规定一些变更是被禁止的，因为它们会导致原来使用超类实例的已有客户端代码在切换至子类时必须要做更改。

1. 前置条件不能被加强

当子类重写包含前置条件的超类方法时，它绝不应该加强现有的前置条件。这样做很可能会影响到那些已经假设超类为所有方法定义了最严格的前置条件契约的客户端代码。

代码清单7-7显示添加了新的`WorldWideShippingStrategy`。由于它与`ShippingStrategy`类很相似，因此将该类实现为后者的子类。`CalcuteShippingCost`方法被重写后，多了一个新防卫子句，作为对`regionInfo`参数代表的包裹目的地信息进行检查。`ShippingStrategy`类并没有要求必须提供包裹目的地信息，但是子类`WorldWideShippingStrategy`则要求必须提供有效的包裹目的地参数，否则它就无法正确计算出包裹运输到目的地的费用。

代码清单7-7　子类通过增加一个新的防卫子句增强了前置条件

```
public class WorldWideShippingStrategy : ShippingStrategy
{
    public override decimal CalculateShippingCost(
        float packageWeightInKilograms,
        Size<float> packageDimensionsInInches,
        RegionInfo destination)
    {
        if (packageWeightInKilograms <= 0f)
            throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package
  weight must be positive and non-zero");

        if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
            throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
  dimensions must be positive and non-zero");

        if (destination == null)
            throw new ArgumentNullException("destination", "Destination must be
  provided");

        return decimal.One;
    }
}
```

加强前置条件的尝试让你能够保证得到有效的目的地参数，但这会引起一个已有的调用代码无法解决的问题。如果某个类调用了ShippingStrategy类的CalculateShippingCost方法，它是可以向目的地参数传入空值且不担心有副作用产生。但是，如果改为调用WorldWideShippingStrategy类的CalculateShippingCost方法后，它就必须要确保传给目的地参数的值不为空。如果传入的是空值，就会违背相应的前置条件要求并且会引发一个异常。正如前面几章所讲述的，客户端代码绝不应该假设类的具体行为。那样做只会在客户端代码和类之间引入紧密耦合关系，从而导致缺乏响应需求变更的能力。

代码清单7-8展示的单元测试可以证明这种问题。

代码清单7-8 当加强前置条件时，客户端无法在需要ShippingStrategy的情况下可靠地使用WorldWideShippingStrategy

```
[Test]
public void ShippingRegionMustBeProvided()
{
    strategy.Invoking(s => s.CalculateShippingCost(1f, ValidDimensions, null))
        .ShouldThrow<ArgumentNullException>("Destination must be provided")
        .And.ParamName.Should().Be("destination");
}
```

如果该测试使用的strategy对象是WorldWideShippingStrategy类的实例，测试的结果会是成功的；没有提供必要的目的地信息，因此会按照期望引发一个异常。相反，如果使用的是ShippingStrategy类实例，该测试将会失败，因为ShippingStrategy类的方法实现中并没有前置条件来检查目的地值是否为空，也不会按照测试期望的那样在检查到空值时引发异常。

代码清单7-9展示了一组重构过的单元测试，它们并不尝试对这两种不同的类测试相同的前置条件。第一个测试只在WorldWideShippingStrategy类的实例上断言必须提供目的地信息。然而，无论是什么类型的运输策略，要求运输重量必须大于零的前置条件都总是有效的，所以作为基类的第二个测试对每种运输策略类型都有效。

代码清单7-9 重构后的单元测试分别针对这两种不同的运输策略类型

```
[TestFixture]
public class WorldWideShippingStrategyTests : ShippingStrategyTestsBase
{
    [Test]
    public void ShippingRegionMustBeProvided()
    {
        strategy.Invoking(s => s.CalculateShippingCost(1f, ValidSize, null))
            .ShouldThrow<ArgumentNullException>("Destination must be provided")
            .And.ParamName.Should().Be("destination");
    }

    protected override ShippingStrategy CreateShippingStrategy()
    {
        return new WorldWideShippingStrategy(decimal.One);
```

```
        }
    }
    // . . .
    public abstract class ShippingStrategyTestsBase
    {
        [Test]
        public void ShippingWeightMustBePositive()
        {
            strategy.Invoking(s => s.CalculateShippingCost(-1f, ValidSize, null))
                .ShouldThrow<ArgumentOutOfRangeException>("Package weight must be positive and
      non-zero")
                .And.ParamName.Should().Be("packageWeightInKilograms");
        }
    }
```

2. 后置条件不能被削弱

在向子类应用后置条件时，规则恰好相反。不是不能加强后置条件，而是不能削弱它们。对于所有与和契约相关的Liskov替换规则而言，你不能够削弱后置条件的原因很明显，因为已有的客户端代码在从原有的超类切换至新的子类时很可能会出错。理论上，如果严格遵守了Liskov替换原则，你所创建的任何子类都能够被所有已有的客户端代码使用且不会引起意料之外的错误。

代码清单7-10展示了在一个已有的客户端代码中引入意外失败的示例，其中包括了与`WorldWideShippingStrategy`类相关的单元测试和实现，该类用来对国际包裹运输建模。

代码清单7-10　新的实现要求削弱后置条件

```
    [Test]
    public void ShippingDomesticallyIsFree()
    {
       strategy.CalculateShippingCost(1f, ValidDimensions, RegionInfo.CurrentRegion)
            .Should().Be(decimal.Zero);
    }
    // . . .
    public override decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
      packageDimensionsInInches, RegionInfo destination)
    {
        if (destination == null)
            throw new ArgumentNullException("destination", "Destination must be provided");

        if (packageWeightInKilograms <= 0f)
            throw new ArgumentOutOfRangeException("packageWeightInKilograms", "Package weight
      must be positive and non-zero");

        if (packageDimensionsInInches.X <= 0f || packageDimensionsInInches.Y <= 0f)
            throw new ArgumentOutOfRangeException("packageDimensionsInInches", "Package
      dimensions must be positive and non-zero");

        var shippingCost = decimal.One;

        if(destination == RegionInfo.CurrentRegion)
        {
            shippingCost = decimal.Zero;
```

```
        }

        return shippingCost;
    }
```

示例中的单元测试用来在当前区域用作目的地（也就是本地运输）时断言WorldWideShippingStrategy类不会对此次运输收费。这也在方法尾部的实现中有所体现。这个新测试再一次与基类的单元测试发生了冲突，原有的测试断言是：方法的返回值必须大于零。如代码清单7-11所示。

代码清单7-11 原有的单元测试会在使用WorldWideShippingStrategy类实例时失败

```
    [Test]
    public void ShippingCostMustBePositiveAndNonZero()
    {
        strategy.CalculateShippingCost(1f, ValidDimensions, RegionInfo.CurrentRegion)
            .Should().BeGreaterThan(0m);
    }
```

这样的改动很容易影响已经对运输费用值有所假设的客户端代码。比如，有客户端已经根据ShippingStrategy类的前置条件契约假设了运输成本总是大于零的，然后该客户端会在后续的计算中使用该运输成本。当切换至新的WorldWideShippingStrategy类时，该客户端却突然开始不断地对所有本地订单引发DivideByZeroException异常。

如果你遵守Liskov替换原则并且决不削弱后置条件，就不会引入这个缺陷。

7

3. 数据不变式必须被保持

在创建新的子类时，它必须继续遵守基类中的所有数据不变式。这里很容易出问题，因为子类有很多机会来改变基类中的私有数据。

代码清单7-12会回到本章前面使用的数据不变式示例。不同的是，这里的示例中，ShippingSrategy类通过构造函数参数接受比例税率数值并且将其保存到一个只读的数据不变式中。新引入的WorldWideShippingStrategy类将比例税率改为一个公共的属性。

代码清单7-12 子类破坏了超类的数据不变式，因此也违背了Liskov替换原则

```
    [Test]
    public void ShippingFlatRateCanBeChanged()
    {
        strategy.FlatRate = decimal.MinusOne;

        strategy.FlatRate.Should().Be(decimal.MinusOne);
    }
    // . . .
    public class WorldWideShippingStrategy : ShippingStrategy
    {
        public WorldWideShippingStrategy(decimal flatRate)
            : base(flatRate)
        {
```

```
    }

    public decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            flatRate = value;
        }
    }
}
```

尽管子类重用了基类的构造函数及其防卫子句，但它并没有保护好原有的数据不变式，因此也违背了Liskov替换原则。上面示例中的单元测试可以证明客户端能够设置负数值。如果该类能够正确地保护原有的数据不变式，就不应该允许将比例税率设置为负数值。

代码清单7-13中，重构后的基类不再允许直接修改比例税率字段，它的子类也正确地保持了比例税率属性这个数据不变式。这种模式非常普遍：私有的字段有对应的受保护的或公共的属性，属性的设置器中包含的防卫子句用来保护属性相关的数据不变式。

代码清单7-13　基类只允许子类通过包括防卫子句的设置器来修改比例税率字段

```
public class WorldWideShippingStrategy : ShippingStrategy
{
    public WorldWideShippingStrategy(decimal flatRate)
        : base(flatRate)
    {

    }

    public new decimal FlatRate
    {
        get
        {
            return base.FlatRate;
        }
        set
        {
            base.FlatRate = value;
        }
    }
}
// . . .
public class ShippingStrategy
{

    public ShippingStrategy(decimal flatRate)
    {
```

```
        if (flatRate <= decimal.Zero)
            throw new ArgumentOutOfRangeException("flatRate", "Flat rate must be positive
  and non-zero");

        this.flatRate = flatRate;
    }

    protected decimal FlatRate
    {
        get
        {
            return flatRate;
        }
        set
        {
            if (value <= decimal.Zero)
                throw new ArgumentOutOfRangeException("value", "Flat rate must be positive
  and non-zero");

            flatRate = value;
        }
    }
}
```

在严格控制了字段的可见性并只允许通过引入防卫子句的属性设置器访问该字段后，该属性相关的数据不变式得到了保护。对子类层次来说，这种方式也是值得推荐的，因为这意味着将来所有的子类都不再需要防卫子句检查，它们无法直接改写超类中的这个字段。

代码清单7-14展示了一个能够断言这个新行为的单元测试。

7

代码清单7-14 在维护了不变式的情况下，此单元测试通过

```
[Test]
public void ShippingFlatRateCannotBeSetToNegativeNumber()
{
    strategy.Invoking(s => s.FlatRate = decimal.MinusOne)
        .ShouldThrow<ArgumentOutOfRangeException>("Flat rate must be positive and non-
  zero")
        .And.ParamName.Should().Be("value");
}
```

如果一个客户端尝试将`FlatRate`属性设置为负数值或零，设置器中的临界子句会阻止赋值的动作并且引发一个`ArgumentOutOfRangeException`异常。

7.2.5 代码契约

上一节通篇都在讲解如何使用防卫子句实现最基本的契约。但是由于使用`if`语句和异常手动构造的防卫子句有些冗长，因此，可以替代临界子句的代码契约是非常值得探讨的，它是一种更好的契约实现方式。

在NET Framework4.0之前，.NET的代码契约特性被组织在一个独立的库中，直到4.0才被集

成到了主库mscorlib.dll中。除了比手动临界子句更易读写和理解外，代码契约还提供了静态验证以及自动生成参考文档的特性。

通过静态契约验证，代码契约能够在不运行应用程序的情况下检查契约的履行情况。这种方式有助于公开空引用和数组越界等隐式契约以及本节中要讲解的各种显式编码契约。

生成方法或者类相关的参考文档非常重要，因为这是客户端能够了解契约期望信息的唯一途径。如果方法和类的参考文档包含了足够的信息，那么客户端就可以通过Visual Studio提供的智能感知特性浏览到这些信息。这能让应用契约的类型更易用一些。

1. 前置条件

使用代码契约可以让前置条件代码变得很简洁。在引用mscorlib.dll程序集中的System.Diagnostics.Contracts命名空间后，你就不再需要引用其他的程序集了。其中的Contract静态类提供了实现契约所需的主要功能。

> **注意** 如果你决定好采用代码契约这种方式，代码中会有很多地方使用Contract静态类。这倒不是个大问题，因为代码契约是一种普遍应用的代码基础结构，通常也不会认为要移除或者替换它。但是，一旦应用之后再想剔除代码契约，工作量就会非常巨大，所以最好在一开始就做好决定：要么全面应用，要么根本不用。

代码清单7-15展示了代码契约前置条件的声明方式。

代码清单7-15 System.Diagnostics.Contracts命名空间能够为方法提供防卫子句

```
using System.Diagnostics.Contracts;

public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires(packageWeightInKilograms > 0f);
        Contract.Requires(packageDimensionsInInches.X > 0f && packageDimensionsInInches.Y
  > 0f);

        return decimal.MinusOne;
    }
}
```

Contract.Requires方法接受一个布尔谓词值。该谓词表示方法主体逻辑执行前需要的状态。注意，这与手动防卫子句的if语句中的谓词恰好相反。手动临界子句会在状态无效时引发异常。而在代码契约中，谓词更接近于单元测试中的断言：布尔条件式的值必须为真，否则就违背了契约。上面的示例要求packageWeightInKilograms参数必须大于零，packageDimensionsInInches参数的X和Y属性都必须大于零。

Contract.Requires方法会在契约谓词没有得到满足时引发一个异常，只是这里的异常类为

ContractException，这与前面示例中已有单元测试期望的异常类不相符。因此，那些已有的单元测试会失败。

```
Expected System.ArgumentOutOfRangeException because Package dimension must be positive and non-
  zero, but found System.Diagnostics.Contracts.__ContractsRuntime+ContractException with message
  "Precondition failed: packageDimensionsInInches.X > 0f && packageDimensionsInInches.Y > 0f"
```

此外，如果在运行示例代码时传入无效参数，你会看到如图7-2所示的消息窗口。这些信息通知你在使用该方法前应该首先配置好代码契约。

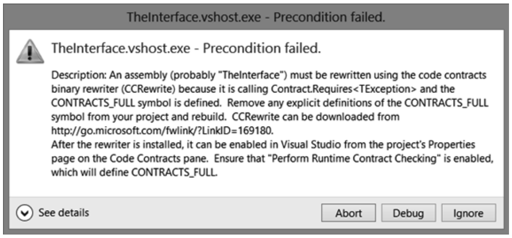


图7-2　在使用代码契约前必须做好配置

Visual Studio中的每个项目属性都包含一个代码契约的标签页，通过它可以配置项目代码契约相关的设置。最小可工作的设置如图7-3所示。

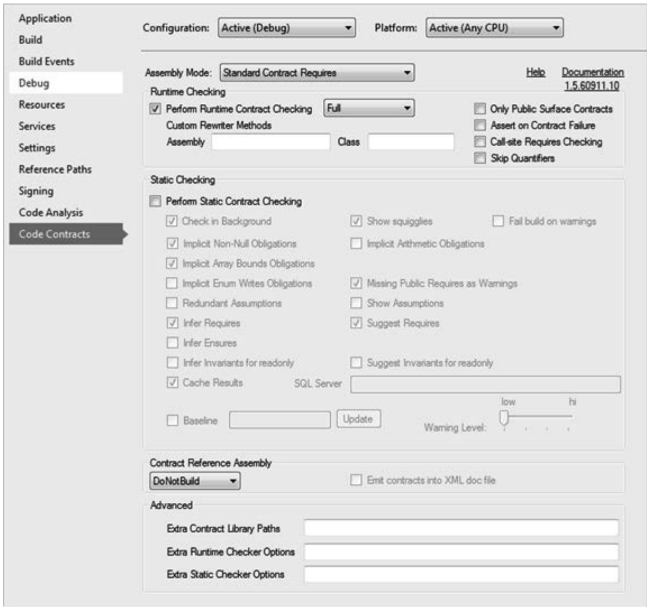


图7-3　代码契约的属性页面包含了很多设置项①

① 要包含这个代码契约的配置标签页，需要下载插件，网址为：https://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970。——译者注

正确配置后，就可以使用另外一个版本的Contract.Requires方法来编写前置条件契约了。如代码清单7-16所示。

代码清单7-16　这个版本的Requires方法可以接受指定要引发的异常类型

```
public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");

        return decimal.MinusOne;
    }
}
```

泛型Requires方法能够接受在谓词不为真时你想要引发的异常类型。上面示例中指定了要引发的异常类型以及第二个参数中的异常信息，这样就应该可以通过已有单元测试的断言了。

2. 后置条件

与前置条件类似，代码契约也为定义后置条件提供了快捷方法。Contract静态类包含的Ensures方法就是用来实现后置条件的。该方法同样是接受一个必须为真的谓词，以便继续执行到方法出口。值得注意的是，Contract.Ensures方法后不可以有返回语句之外的其他语句。作这样的要求是很有意义的，因为任何返回语句之外的后续代码都有可能破坏后置条件相关的状态。

代码清单7-17展示了ShippingCostMustBePositive单元测试以及使用Contract.Ensures方法重构后置条件实现后的CalculateShippingCost方法。

代码清单7-17　使用Ensures方法创建的后置条件在方法出口时应该为真

```
[Test]
public void ShippingCostMustBePositive()
{
    strategy.CalculateShippingCost(1, ValidSize, null)
        .Should().BeGreaterThan(decimal.MinusOne);
}
// . . .
public class ShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");
```

```
            Contract.Ensures(Contract.Result<decimal>() > 0m);

            return decimal.MinusOne;
        }
    }
```

这个示例中的谓词看起来与前面示例中的谓词不同，也与常见的判断返回值是否有效的后置条件使用方式不同。检查运输费用是否大于零（实际上是非负数）需要明白返回值的相关信息。返回值通常会，但并不总是会，声明和定义在方法内部的局部变量。你可以谨慎地断言返回的值大于零，但是这并不像你想象的那样简单。为了获取从方法实际返回的值，你需要使用Contract.Result方法。这个泛型方法可以接受方法的返回值类型并返回方法最终退出时刻的任意结果。通过Contract.Result方法获取契约所在方法的最终返回值，你就能确保在后置条件语句后没有任何代码能在不引起失败异常的情况下将返回值替换为无效值。

3. 数据不变式

通常类中的每个方法都包含自己的前置条件和后置条件，但数据不变式是针对类整体的契约。代码契约允许你在类内创建一个私有的方法来声明和定义针对类整体的所有数据不变式。

如代码清单7-18所示，数据不变式可以由Contract静态类的另外一个方法进行定义。

代码清单7-18 有专用的方法可以用来保护数据不变式

```
public class ShippingStrategy
{
    public ShippingStrategy(decimal flatRate)
    {
        this.flatRate = flatRate;
    }

    [ContractInvariantMethod]
    private void ClassInvariant()
    {
        Contract.Invariant(this.flatRate > 0m, "Flat rate must be positive and non-zero");
    }

    protected decimal flatRate;
}
```

Contract类的Invariant方法和Requires以及Ensures方法的使用模式一致，都是接受一个为满足契约而必须为真的布尔谓词。上面的示例中，Invariant方法还有第二个字符串参数用来描述数据不变式未被保护而导致契约失败的错误信息。我们鼓励尽可能多地使用Invariant方法来保护每个数据不变式，最好把不同目的的数据不变式分隔开，也就是说，不要用逻辑AND运算符&&把它们串在一起。这样做的好处就是你在失败时能够准确地知道是哪个数据不变式被破坏了。

如果ClassInvariant方法只是一个普通的私有方法，你就需要亲自在每个方法出入口调用

该方法以确保正确保护了所有的数据不变式。幸运的是，代码契约给你提供了更好的方式：只需要使用ContractInvariantMethodAttribute来标记方法即可。注意，属性并不需要Attribute后缀，因此这里把属性名简化为ContractInvariantMethod。代码契约会要求类中的其他方法的出入口处必须调用带有这个标记的方法以确保该类的所有数据不变式没有被破坏。

可以应用ContractInvariantMethod标记的方法必须满足既没有返回值也没有参数的前提条件，当然，你仍然可以决定方法的名称和访问级别，也就是说，可以根据需要为它起名，也可以把它的访问级别设置为公共的或者私有的。一个类可以有多个ContractInvariantMethod标记的方法，因此可以在逻辑上将不同目的的数据不变式组织在独立的方法中。最后要强调的是，这些带有ContractInvariantMethod标记的方法内必须只包含对Contract.Invariant方法的调用。

4. 接口契约

这里要讲述的最后一个代码契约的特性就是接口契约。到目前为止，你都是在类的实现上应用Contract静态类的Requires、Ensures和Invariant方法。前面也有提到，Contract类的静态本质使得对其方法的调用遍布代码的各个角落，从而导致后期移除或者用其他库替换它们时会变得非常困难。这在一定程度上破坏了自适应代码的理想状态，但是在基础结构上由于实际原因而作的妥协是可以理解的。

最急迫的是由于在类实现上不加限制地应用代码契约而导致代码可读性急剧下降的问题。实际上，这并不真的只是.NET代码契约特性实现的错，持续应用任何契约实现都会造成同样的问题。无论用什么方式在代码中实现前置条件、后置条件和数据不变式，有效代码率都会降低。

接口契约能够解决这些问题并提供另外一个很有用的特性。代码清单7-19展示了ShoppingStrategy类的接口契约示例。

代码清单7-19　使用专用类为接口的所有实现定义前置条件、后置条件和数据不变式

```
[ContractClass(typeof(ShippingStrategyContract))]
interface IShippingStrategy
{
    decimal CalculateShippingCost(
        float packageWeightInKilograms,
        Size<float> packageDimensionsInInches,
        RegionInfo destination);
}
//. . .
[ContractClassFor(typeof(IShippingStrategy))]
public abstract class ShippingStrategyContract : IShippingStrategy
{
    public decimal CalculateShippingCost(float packageWeightInKilograms, Size<float>
  packageDimensionsInInches, RegionInfo destination)
    {
        Contract.Requires<ArgumentOutOfRangeException>(packageWeightInKilograms > 0f,
  "Package weight must be positive and non-zero");
        Contract.Requires<ArgumentOutOfRangeException>(packageDimensionsInInches.X > 0f &&
  packageDimensionsInInches.Y > 0f, "Package dimensions must be positive and non-zero");
```

```
            Contract.Ensures(Contract.Result<decimal>() > 0m);

            return decimal.One;
        }

        [ContractInvariantMethod]
        private void ClassInvariant()
        {
            Contract.Invariant(flatRate > 0m, "Flat rate must be positive and non-zero");
        }
    }
```

当然，接口契约要先有应用契约的目标接口。上面示例中，CalculateShippingCost方法已经被提取到自己所属的IShippingStrategy接口中了。该示例是对该接口应用契约，不再只是针对单个类实现应用契约。这一点很重要，它让这个示例与前面所有示例有了质的区别，因为它对所有该接口的实现类都有效。这就是你能通过一些实现和指令增强接口的方式，而增强后的接口能拥有更强大的需求和保障。

在编写接口契约代码时，你还需要一个专门的类来实现接口的方法，其中的方法体只包含对Contract静态类的Requires、Ensures的调用。抽象ShippingStrategyContract类提供的功能实现看起来与前面示例中的具体类相似，只是方法中缺少了实际的功能实现代码。即使在产品代码中，契约类中包含的代码也是这样的。与真正的实现类一样，也有一个带有Contract-InvariantMethod标记的方法，其中只包含所有对Contract静态类的Invariant方法的调用。

不幸的是，为了关联接口和契约类的实现，你需要通过一个属性完成一次双向引用。这种方式不够好，因为它会对接口的简洁性造成影响，如果能避免会更好。尽管如此，使用ContractClass和ContractClassFor属性分别对接口和契约类进行标记后，你只需要实现一次前置条件、后置条件和数据不变式的防卫代码，而它们对该接口上的所有实现类都是效的。ContractClass和ContractClassFor属性都可以接受模板类，前者用于标记接口并需要传入契约类，而后者用于标记契约类并需要传入接口类。

到此，我们要结束对代码契约的介绍了，让我们重新回到Liskov替换原则中与契约相关的主题讨论上。最后需要特别强调的一点是：无论是手动编码还是使用代码契约实现，如果契约中有任何前置条件、后置条件或者数据不变式的断言失败了，客户端都不应该再捕获到代码异常。对于客户端而言，捕获异常这个行为本身代表了它还能从捕获的异常状态中恢复，但是契约被破坏后通常都不会也不太可能被恢复。最理想的情况就是通过功能测试发现了所有违背契约的问题，并在交付产品之前修复所有这些发现的问题。因此，对契约做单元测试就变得非常重要了。如果在交付产品代码后发现有个破坏契约的问题依然存在且很不幸地被用户发现时，最好的处理方式很可能就是强制关闭应用程序。这时让应用程序以失败的方式退出是恰当的，因为此时应用程序的内部状态很可能是无效且无法恢复的。对于网页应用，这意味着显示一个全局的错误页面。对于桌面应用，可以给用户一个友善的提示并能让他们有机会报告出现的问题。所有情况下，都应该使用日志记录发生的异常、当时完整的堆栈跟踪信息以及其他尽可能多的上下文信息。

下一节会集中讲解Liskov替换原则剩余的规则，它们主要应用在协变和逆变上。

7.3　协变和逆变

Liskov替换原则的剩余规则都与协变和逆变相关。通常来说，变体（variance）这个术语主要应用于复杂类层次结构中以定义子类型的期望行为。

7.3.1　定义

与前面几节类似，在详细讲解Liskov替换原则在变体上的需求前，会首先讲解变体相关的基本概念。

1. 协变

图7-4展示了一个非常小的类层次结构，它只包含了两个根据分类命名的类型：`Supertype`和`Subtype`。`Supertype`定义了被`Subtype`继承的字段和方法。`Subtype`通过定义自己的字段和方法来增强`Supertype`。

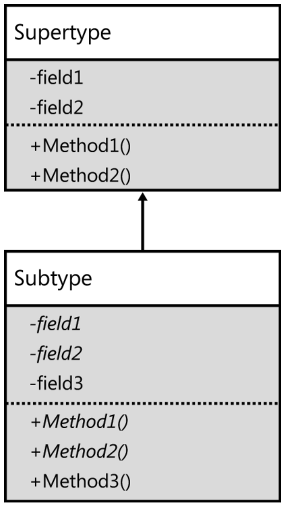


图7-4　在这个类层次结构中，`Supertype`和`Subtype`是父子关系

多态（polymorphism）是一种子类型被看作超类型实例的能力。我们要感谢多态这个面向对象编程的强大特性（C#也支持）。任何能够接受`Supertype`类型实例的方法也可以直接接受`Subtype`类型实例，客户端或服务代码都不需要做类型转换，服务也不需要知道任何子类型相关的信息。服务代码根本就不关心具体是哪个子类型，它们只知道自己处理的是个`Supertype`类型的实例。

此时，再引入另外一个通过泛型参数使用`Supertype`和/或`Subtype`的类型时，我们就需要讨论变体相关的主题了。

图7-5是协变（covariance）概念的一个形象的解释。

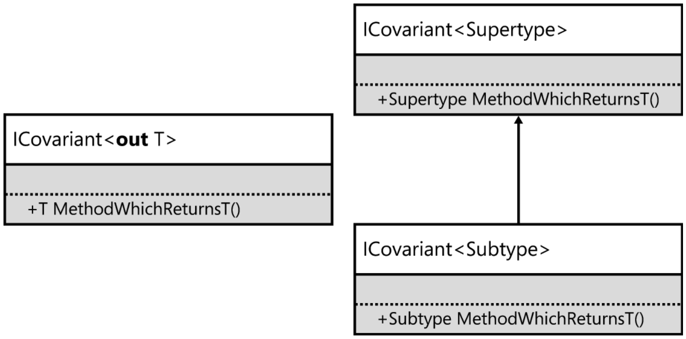


图7-5 由于泛型参数的协变，基类/子类关系被保留下来

很有意思的是，这和前面一样也用到了多态这个强大的特性。因为有了协变，当有方法需要ICovariant<Supertype>的实例时，你完全可以放心地使用ICovariant<Subtype>的实例替代它。能取得这样的效果要归功于协变和多态的紧密配合。

到目前为止，我采用的都是通用的示例。为了巩固对协变概念的解释，我将会用一个更真实的场景来替代前面的类图以及其中示意性的名称。代码清单7-20展示了通用的Entity基类和具体的User子类间的类层次结构。所有Entity类型都有一个GUID标识符和一个字符串名称，而每个User类都有一个EmailAddress属性和一个DateOfBirth属性。

代码清单7-20 在这个小域中，User就是一个特殊的Entity类型

```
public class Entity
{
    public Guid ID { get; private set; }

    public string Name { get; private set; }
}
// . . .
public class User : Entity
{
    public string EmailAddress { get; private set; }

    public DateTime DateOfBirth { get; private set; }
}
```

这个示例与Supertype/Subtype示例很类似，但是目的性更强。在这个小小的问题域中需要应用存储库模式。存储库模式会提供一个获取对象的接口，这些对象看起来是在内存中，但是实际上很有可能是从某个完全不同的存储介质中加载进来的。代码清单7-21展示了Entity-Repository类和它的UserRepository子类。

代码清单7-21 不使用泛型，C#中的所有继承都是非变体

```
public class EntityRepository
{
```

```
    public virtual Entity GetByID(Guid id)
    {
        return new Entity();
    }
}
// . . .
public class UserRepository : EntityRepository
{
    public override User GetByID(Guid id)
    {
        return new User();
    }
}
```

这个示例与前面的有所不同，其中关键的一点是：不使用泛型类型，C#方法的返回类就不是协变的。实际上，在子类中尝试将GetByID方法的返回类型更改为User类会直接引起一个编译错误。

```
error CS0508: 'SubtypeCovariance.UserRepository.GetByID(System.Guid)': return type must be
  'SubtypeCovariance.Entity' to match overridden member
  'SubtypeCovariance.EntityRepository.GetByID(System.Guid)'
```

也许你只是靠经验判断这样的更改是无法工作的，但是真正出错的原因则是因为这种情况下的继承是不具备协变能力的。如果C#在继承时支持普通类上的协变，你就能够在UserRepository类中直接更改方法的返回类型。可惜C#并没有这种能力，所以你只有两个可用的选项。

第一，你可以把UserRepository类的GetByID方法的返回类型修改回Entity类型，然后在该方法返回的地方应用多态将Entity类型的实例转换为User类型的实例。这个方式不够好，因为这要求UserRepository的客户端必须自己做实例类型转换，或者必须针对User类型做探测，如果返回的是User类型，就执行某些针对性的代码。

第二，你还可以把EntityRepository重新定义为一个需要泛型的类型，可以把Entity类型作为泛型参数传入。这个泛型参数是可以协变的，UserRepository子类可以为User类型指定超类型。代码清单7-22举例说明了第二种方式。

代码清单7-22　泛型化的基类可以利用协变能力，从而允许子类重写返回类型

```
public interface IEntityRepository<TEntity>
    where TEntity : Entity
{
    TEntity GetByID(Guid id);
}
// . . .
public class UserRepository : IEntityRepository<User>
{
    public User GetByID(Guid id)
    {
        return new User();
    }
}
```

示例代码中并没有继续使用可以实例化的具体EntityRepository，而是引入了一个没有GetByID方法默认实现的接口。这里使用接口虽然不是必须的，但仍然是很合理的，因为我们前面一直在强调，让客户端代码依赖接口要比依赖实现好很多。

你也会注意到，接口泛型参数后面还有一个where子句。应用where子句的接口比原有实现有了更高的灵活度，因为where子句可以保证接口的子类总是传入Entity类型层次结构中的类型。

新的UserRepository类的客户端无需再做向下的类型转换，因为它们直接得到的就是User类型对象，而不是Entity类型对象，同时，EntityRepository和UserRepository两个类之间的父子继承关系也得以保留。

2. 逆变

逆变是一个类似于协变的概念。协变只是与方法返回类型的处理相关，而逆变（contravariance）是与方法参数类型的处理相关。

图7-6使用前面讨论过的Supertype和Subtype示例，展示了通过泛型参数逆变的类型之间的关系。

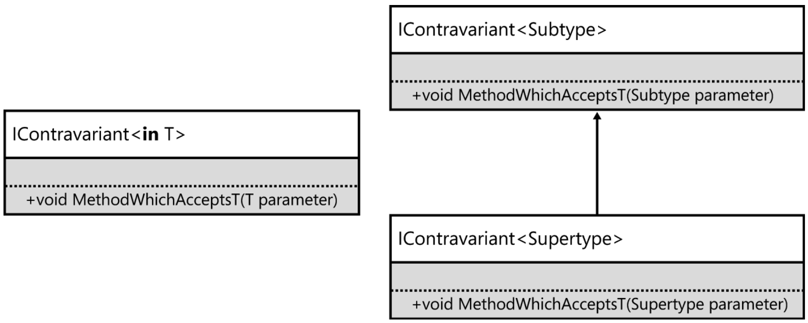


图7-6 由于泛型参数的逆变，基类/子类关系被倒置了

IContravariant接口定义的方法只接受由泛型参数指定类型的单个参数。这里的泛型参数由关键字in标记，表示它是可逆变的。

后续的类层次结构可以以此类推，这表明了继承层次结构已经被颠倒了：IContravariant<Subtype>成为了超类，IContravariant<Supertype>则变成了子类，尽管从直觉上看来有些别扭和奇怪，但后面你很快会知道为什么逆变会有这种行为以及为什么它很有用。

代码清单7-23展示了.NET Framework提供的IEqualityCompare接口的一个具体的实现。EntityEqualityComparer类的Equals方法接受前面定义的Entity类型参数。比较过程的实现并不重要，只是简单地比较了两个实体的身份标识。

代码清单7-23 IEqualityComparer接口允许定义诸如EntityEqualityComparer类这样的功能对象

```
public interface IEqualityComparer<in TEntity>
```

```
    where TEntity : Entity
{
    bool Equals(TEntity left, TEntity right);
}
// . . .
public class EntityEqualityComparer : IEqualityComparer<Entity>
{
    public bool Equals(Entity left, Entity right)
    {
        return left.ID == right.ID;
    }
}
```

代码清单7-24展示了逆变在EntityEqualityComparer类上的作用。

代码清单7-24　逆变颠倒了类层次结构，它允许在需要具体比较器的地方传入更通用的比较器

```
[Test]
public void UserCanBeComparedWithEntityComparer()
{
    SubtypeCovariance.IEqualityComparer<User> entityComparer = new
  EntityEqualityComparer();
    var user1 = new User();
    var user2 = new User();
    entityComparer.Equals(user1, user2)
        .Should().BeFalse();
}
```

如果没有逆变（接口定义中泛型参数前不起眼的in关键字），编译时会直接报错。

```
error CS0266: Cannot implicitly convert type 'SubtypeCovariance.EntityEqualityComparer' to
  'SubtypeCovariance.IEqualityComparer<SubtypeCovariance.User>'. An explicit conversion exists
  (are you missing a cast?)
```

错误信息告诉你，这里并没有从EntityEqualityComparer到IEqualityComparer<User>的类型转换器，直觉上就是这样的，因为Entity是超类型，而User是子类型。然而，因为IEqualityComparer支持逆变，所以现有的继承层次结构被颠倒了，此时你就能够做到通过使用IEqualityComparer接口向需要具体类型参数的地方传入更通用的类型。

3. 不变性

除了协变和逆变的行为外，类型本身具有不变性。这里的不变性与本章前面讲述的代码契约中的数据不变性（data invariant）是不同的。这里的不变性是指“不会生成变体”。如果一个类型完全不能生成变体，那么就无法在该类型上形成类型层次结果。代码清单7-25使用IDictionary泛型类型来说明这个事实。

代码清单7-25　有些泛型类型既不可协变也不可逆变，因此它们具有不变性

```
[TestFixture]
public class DictionaryTests
{
    [Test]
```

```
    public void DictionaryIsInvariant()
    {
        // Attempt covariance...
        IDictionary<Supertype, Supertype> supertypeDictionary = new Dictionary<Subtype,
  Subtype>();

        // Attempt contravariance...
        IDictionary<Subtype, Subtype> subtypeDictionary = new Dictionary<Supertype,
  Supertype>();
    }
}
```

DictionaryIsInvariant测试方法第一行试图在给一个键和值类型都是Supertype的字典赋值，但所赋字典对象的键和值类型却都是Subtype类型的。因为IDictionary类型是不可协变的，因此这一句代码不会成功。

第二行代码也是无效的，因为它在尝试做倒置：给一个键和值类型都是Subtype的字典赋值一个键和值类型都是Supertype类型的字典。失败的原因是因为IDictionary类型也是不可逆变的，因此无法倒置Subtype和Supertype类型间的继承层次结构。

从IDictionary类型既不可协变也不可逆变的事实即可知道它必定是个非变体。的确是这样的，代码清单7-26展示了IDictionary接口的声明方式，你可以看到，定义中并没有对in和out关键字的引用，而这二者则分别是用来指定协变和逆变特性的。

代码清单7-26 IDictionary接口的所有泛型参数都没有in或out关键字标记

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey, TValue>>,
  IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
```

7

与前面证明的一般情况（也就是说，没有泛型类型）一样，C#语言的方法参数类型和返回类型都是不可变的。只有在涉及泛型时才能将类型定义为可协变的或可逆变的。

7.3.2 Liskov 类型系统规则

现在你已经了解了变体的基础，本节会回到Liskov替换原则相关的主题上。Liskov替换原则定义了以下三个规则，其中前两个都是与变体相关的。

- ❑ 子类型的方法参数必须是可逆变的。
- ❑ 子类型的返回类型必须是可协变的。
- ❑ 不允许引发新异常。

只有方法参数支持逆变而且返回类型支持协变，你才可以编写出严格遵守Liskov替换原则的代码。

第三条规则与可变性无关，因此在下面单独讨论一下。

不允许引发新异常

这条规则比其他两条Liskov替换原则规则要直观的多，它主要是与编程语言的类型系统相

关。你首先要思考的是：什么才是异常的真正目的？

异常机制的主旨就是将错误的汇报和处理环节分隔开。通常情况下，汇报器和处理器就是目的和应用场景截然不同的两种类型。异常对象表示通过该异常类型发生的错误以及相应的数据。任何代码都可以捕获和响应异常，同样，任何代码也都可以构造和引发异常。尽管如此，最好还是在代码确定能做一些有意义的处理时才去捕获异常。比如，简单的数据库事务的回滚处理，或是复杂的能让用户看到并反馈详细错误信息给开发人员的绚丽界面的处理。

同样，捕获异常后不做任何处理或只捕获最通用的Exception基类都是不可取的。二者结合的情况就更糟糕了。对于只捕获最通用的Exception基类这种情况，你实际上是在尝试捕获和响应任何异常，包括那些你实际上根本无法处理和恢复的情况，比如OutOfMemoryException、StackOverflowException或ThreadAbortException等。如果要想改善这种状况，你需要确保总是从ApplicationException类派生自己的异常，因为很多无法恢复的异常都是从System-Exception类派生出来的。然而，这要取决于你所依赖的第三方库是否也采用了这种好的实践方式。

代码清单7-27展示的两个异常类型在同一个类层次结构下是兄弟关系。这种兄弟关系可以防止只针对单个异常的捕获代码块却能截获两种异常的情况。

代码清单7-27　两个异常都是Exception类，但是二者并非父子继承关系

```
public class EntityNotFoundException : Exception
{
    public EntityNotFoundException()
        : base()
    {

    }

    public EntityNotFoundException(string message)
        : base(message)
    {

    }
}
//. . .
public class UserNotFoundException : Exception
{
    public UserNotFoundException()
        : base()
    {

    }

    public UserNotFoundException(string message)
        : base(message)
    {

    }
}
```

如果想要在单个捕获代码块中同时捕获EntityNotFoundExcetpion和UserNotFound-Exception两个异常，你应该去捕获通用的Exception类型，但这并不是值得推荐的做法。

代码清单7-28展示的EntityRepository和UserRepository类实现代码会加重这个问题。

代码清单7-28 同一接口的两个不同的实现很有可能引发两种不同类型的异常

```
public Entity GetByID(Guid id)
{
    Contract.Requires<EntityNotFoundException>(id != Guid.Empty);

    return new Entity();
}
//. . .
public User GetByID(Guid id)
{
    Contract.Requires<UserNotFoundException>(id != Guid.Empty);

    return new User();
}
```

这两个类型都使用了代码契约来断言一个前置条件：方法的输入参数id必须不等于Guid.Empty。二者都会在不满足契约时引发自有的异常类型。我们一起来细想一下示例实现代码对使用存储库的客户端的影响。客户端代码需要考虑去捕获这两种异常，但是如果不捕获Exception类型，就无法在单个捕获代码块中同时捕获这两种异常类型。代码清单7-29展示的单元测试是这两个存储库类型的客户端。

代码清单7-29 因为无法将UserNotFoundException赋值给EntityNotFoundException异常，所以单元测试会失败

```
[TestFixture(typeof(EntityRepository), typeof(Entity))]
[TestFixture(typeof(UserRepository), typeof(User))]
public class ExceptionRuleTests<TRepository, TEntity>
    where TRepository : IEntityRepository<TEntity>, new()
{
    [Test]
    public void GetByIDThrowsEntityNotFoundException()
    {
        var repo = new TRepository();
        Action getByID = () => repo.GetByID(Guid.Empty);

        getByID.ShouldThrow<EntityNotFoundException>();
    }
}
```

因为UserRepository并不会像要求的那样引发EntityNotFoundException异常，所以这个单元测试会失败。如果UserNotFoundException是EntityNotFoundException的子类，这个测试就可以成功而且单个捕获代码块也能够保证捕获两种类型。

这个问题会成为客户端维护的一个负担。如果客户端是在使用接口并调用接口的方法，那么它就不应该知道接口之后的任何信息。这就回到了随从反模式和阶梯模式之间的争论。如果引入了不属于期望异常的类层次结构的新异常，客户端就必须直接引用这些新的异常。而且更糟糕的是，客户端将不得不在任何新的异常类型引入时做出相应的更改。

相反，每个接口都应该有一个统一的基础异常类型，它可以将必要的错误信息从异常汇报器传给异常处理器。

7.4　总结

表面上看，Liskov替换原则是SOLID原则中最复杂的一个。你需要理解契约和变体的概念才可以应用Liskov替换原则编写具有更高自适应能力的代码。

默认情况下，接口并不会向用户传达前置条件和后置条件的规则。通过创建防卫子句可以让应用程序在运行时进一步约束参数的有效值范围。Liskov替换原则提出了一些指导原则，比如，子类不能加强前置条件或削弱后置条件等。

相似地，Liskov替换原则也对子类型的可变性提出了一些规则。子类型的方法参数和返回类型应该分别是可逆变的和可协变的。此外，任何可能随着某个新的接口实现而引入的新异常都应该派生一个已有的基础异常类型。如果不这样做，会很可能导致现有的客户端错过捕获这个新的异常类型而导致程序崩溃。

如果没有遵守Liskov替换原则的这些规则，客户端要处理好类层次结构中所有类之间的关系会变得更难。理想情况下，不论运行时使用的是哪个具体的子类型，客户端都可以只引用一个基类或接口而且无需担心行为变化。这么多问题混合在一起会引起代码之间的依赖，因此最好是把它们分隔开。任何对Liskov替换原则定义的规则的违背都应该被看作技术债务，而且像前面几章讲到的，最好能尽早地处理掉这些技术债务，否则后患无穷。

第 8 章

接口分离原则

完成本章学习之后，你将学到以下技能。

- 理解接口分离原则的重要性。
- 主要按照客户端代码的需求编写接口。
- 创建更小更清晰的接口。
- 识别可以应用接口分离原则的场景。
- 按照接口实现的依赖关系分割接口。

基于前面几章的讲解，我们已经知道，在现代面向对象编程的工具集中，接口是一个非常关键的武器。接口所表达的是客户端代码需求和需求具体实现之间的边界。接口分离原则主张接口应该足够小。

接口的每个成员（属性、事件和方法）都需要按照接口的整体目标来实现。除非接口的所有客户端都需要所有成员，否则要求每个实现都满足一个大而全的契约是毫无意义的。要牢记单一职责原则和可以轻易使用的修饰器模式，对于接口包含的每个成员而言，若要实现为修饰器，必须要有一个对应的有效类比。

最简单的接口只有一个服务于单个目的的方法。这种粒度的接口看起来很像是委托，但是它们要比委托强大很多。

8.1 一个分离接口的示例

本章会完整讲解一个从单个巨型接口到多个小型接口分离过程的示例。分离过程中创建了各种各样的修饰器，用于详细讲解大量应用接口分离原则能带来的主要好处。

8.1.1 一个简单的 CRUD 接口

下面这个接口本身很简单，只有五个方法。它用于允许用户对实体对象的持久存储进行CRUD操作。CRUD代表创建、读取、更新和删除（create, read, update, and delete）。这四个动作是客户端维护实体对象持久存储的最常见的一组操作。图8-1展示的UML类图给出了`ICreateReadUpdateDelete`接口上的可用操作。

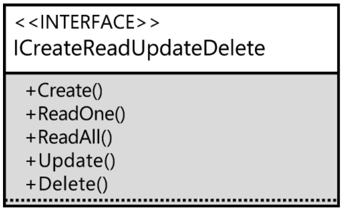


图8-1　最初的还未分离的接口

读取操作被分割成为两个方法，一个用于从存储中获取单个记录，另外一个用于一次获取所有记录。代码清单8-1展示了该接口的代码。

代码清单8-1　一个用于对实体对象做CRUD操作的简单接口

```
public interface ICreateReadUpdateDelete<TEntity>
{
    void Create(TEntity entity);

    TEntity ReadOne(Guid identity);

    IEnumerable<TEntity> ReadAll();

    void Update(TEntity entity);

    void Delete(TEntity entity);
}
```

`ICreateReadUpdateDelete`是一个泛型接口，可以接受不同的实体类型。然而，这种泛型化接口而不是泛型化每个方法的方式，需要客户端首先声明自己要依赖的`TEntity`。如果客户端想要对多种类型的实体做CRUD操作，就需要为每个实体类型创建一个`ICreateReadUpdate-Delete<TEntity>`实例。

> **注意**　尽管客户端需要为每个实体类型创建一个`ICreateReadUpdateDelete<TEntity>`接口实例，但是只需要`ICreateReadUpdateDelete<TEntity>`的一个实现就能够满足所有具体的`TEntity`类型。因为这个实现也是泛型化的。

CRUD中的每个操作都是由对应的`ICreateReadUPdateDelete`接口实现来执行，也包括所有修饰器实现。如代码清单8-2中展示的日志和事务处理等修饰器实现都是可以接受的。

代码清单8-2　有些修饰器作用于所有方法

```
public class CrudLogging<TEntity> : ICreateReadUpdateDelete<TEntity>
{
    private readonly ICreateReadUpdateDelete<TEntity> decoratedCrud;
    private readonly ILog log;
    public CrudLogging(ICreateReadUpdateDelete<TEntity> decoratedCrud, ILog log)
```

```
    {
        this.decoratedCrud = decoratedCrud;
        this.log = log;
    }

    public void Create(TEntity entity)
    {
        log.InfoFormat("Creating entity of type {0}", typeof(TEntity).Name);
        decoratedCrud.Create(entity);
    }

    public TEntity ReadOne(Guid identity)
    {
        log.InfoFormat("Reading entity of type {0} with identity {1}",
  typeof(TEntity).Name, identity);
        return decoratedCrud.ReadOne(identity);
    }

    public IEnumerable<TEntity> ReadAll()
    {
        log.InfoFormat("Reading all entities of type {0}", typeof(TEntity).Name);
        return decoratedCrud.ReadAll();
    }

    public void Update(TEntity entity)
    {
        log.InfoFormat("Updating entity of type {0}", typeof(TEntity).Name);
        decoratedCrud.Update(entity);
    }

    public void Delete(TEntity entity)
    {
        log.InfoFormat("Deleting entity of type {0}", typeof(TEntity).Name);
        decoratedCrud.Delete(entity);
    }

}
// . . .
public class CrudTransactional<TEntity> : ICreateReadUpdateDelete<TEntity>
{
    private readonly ICreateReadUpdateDelete<TEntity> decoratedCrud;
    public CrudTransactional(ICreateReadUpdateDelete<TEntity> decoratedCrud)
    {
        this.decoratedCrud = decoratedCrud;
    }

    public void Create(TEntity entity)
    {
        using (var transaction = new TransactionScope())
        {
            decoratedCrud.Create(entity);

            transaction.Complete();
        }
```

8

```
    }

    public TEntity ReadOne(Guid identity)
    {
        TEntity entity;
        using (var transaction = new TransactionScope())
        {
            entity = decoratedCrud.ReadOne(identity);

            transaction.Complete();
        }
        return entity;
    }

    public IEnumerable<TEntity> ReadAll()
    {
        IEnumerable<TEntity> allEntities;
        using (var transaction = new TransactionScope())
        {
            allEntities = decoratedCrud.ReadAll();

            transaction.Complete();
        }
        return allEntities;
    }

    public void Update(TEntity entity)
    {
        using (var transaction = new TransactionScope())
        {
            decoratedCrud.Update(entity);

            transaction.Complete();
        }
    }

    public void Delete(TEntity entity)
    {
        using (var transaction = new TransactionScope())
        {
            decoratedCrud.Delete(entity);

            transaction.Complete();
        }
    }

}
```

用于记录日志和管理事务的修饰器都属于横切关注点（cross-cutting concern）。几乎所有的接口以及接口中的方法都可以应用日志和事务管理这两个修饰器。因此，为了避免在多个接口中重复实现，你可以使用面向方面编程来修饰接口的所有实现。

有些修饰器只应用于接口的部分方法上，而不是所有的方法。比如，你也许想要在从持久存储中永久删除某个实体前提示用户，这也是一个很常见的需求。切记，请不要去改变现有的类实现，这会违背开放与封闭原则。相反，你应该创建客户端用来执行删除动作接口的一个新实现。代码清单8-3展示了`ICreateReadUpdateDelete<TEntity>`接口的`Delete`方法。

代码清单8-3　在只要求修饰部分接口时可以使用接口分离

```
public class DeleteConfirmation<TEntity> : ICrud<TEntity>
{
    private readonly ICreateReadUpdateDelete<TEntity> decoratedCrud;
    public DeleteConfirmation(ICreateReadUpdateDelete<TEntity> decoratedCrud)
    {
        this.decoratedCrud = decoratedCrud;
    }

    public void Create(TEntity entity)
    {
        decoratedCrud.Create(entity);
    }

    public TEntity ReadOne(Guid identity)
    {
        return decoratedCrud.ReadOne(identity);
    }

    public IEnumerable<TEntity> ReadAll()
    {
        return decoratedCrud.ReadAll();
    }

    public void Update(TEntity entity)
    {
        decoratedCrud.Update(entity);
    }

    public void Delete(TEntity entity)
    {
        Console.WriteLine("Are you sure you want to delete the entity? [y/N]");
        var keyInfo = Console.ReadKey();
        if (keyInfo.Key == ConsoleKey.Y)
        {
            decoratedCrud.Delete(entity);
        }
    }
}
```

顾名思义，`DeleteConfirmation<TEntity>`类只修饰了`Delete`方法。其余方法都是直托方法。直托（pass-through）代表对该方法不做任何实际的修饰：不修饰接口方法，就好像直接调用被修饰的接口方法一样。尽管实际上这些直托方法什么都没有做，为了确保单元测试的覆盖率并确认它们是否正确委托，依然需要为这些直托方法编写测试方法以验证方法行为是否正确。但这

样做与接口分离的方式比较起来要麻烦很多。

通过把Delete方法从ICreateReadUpdateDelete<TEntity>接口分离后，你会得到以下两个接口，如代码清单8-4所示。

代码清单8-4　ICreateReadUpdateDelete接口被一分为二

```
public interface ICreateReadUpdate<TEntity>
{
    void Create(TEntity entity);

    TEntity ReadOne(Guid identity);

    IEnumerable<TEntity> ReadAll();

    void Update(TEntity entity);
}
// . . .
public interface IDelete<TEntity>
{
    void Delete(TEntity entity);
}
```

将ICreateReadUpdateDelete接口一分为二后,就可以只对IDelete<TEntity>接口提供确认修饰器的实现，如代码清单8-5所示。

代码清单8-5　只在相关的接口上应用确认修饰器

```
public class DeleteConfirmation<TEntity> : IDelete<TEntity>
{
    private readonly IDelete<TEntity> decoratedDelete;

    public DeleteConfirmation(IDelete<TEntity> decoratedDelete)
    {
        this.decoratedDelete = decoratedDelete;
    }

    public void Delete(TEntity entity)
    {
        Console.WriteLine("Are you sure you want to delete the entity? [y/N]");
        var keyInfo = Console.ReadKey();
        if (keyInfo.Key == ConsoleKey.Y)
        {
            decoratedDelete.Delete(entity);
        }
    }
}
```

上面示例中有了一些改进，代码量更少了，代码意图也变得更清晰了，不再需要那么多直托方法。同样，代码量变少也意味着相应的测试工作量也变少了。

在讨论下一个修饰器前，先考虑是否能重构现有的DeleteConfirmation修饰器。你应该把要求用户答复的动作封装在一个简单的接口中，这样就可以为多种用户交互类型（比如，控制台、Windows Forms和Windows Presentation Foundation等）编写对应的接口实现，同时也不会影响该接口的修饰器。因为DeleteConfirmation类现在并不符合单一职责原则，所以需要重构它。重构原有DeleteConfirmation类的具体理由有两个：该类委托的接口已经发生了改变，以及会有不同的获得用户答复的交互方式。要求用户确认是否真的想要删除某个实体只需要一个非常简单的类似于谓词的接口，正如代码清单8-6所示。

代码清单8-6　一个很简单的接口，用于征求用户对某些事情的确认

```
public interface IUserInteraction
{
    bool Confirm(string message);
}
```

8.1.2　缓存

下一个需要你实现的修饰器是针对ReadOne和ReadAll这两个读取方法的。你想在这两个读取方法中缓存读取的数据并用作后续请求的返回。而对于Create和Update方法而言，缓存都是没有意义的，代码清单8-7中的第一个修饰器包含了并不需要的直托方法。

代码清单8-7　缓存修饰器包含了冗余的直托方法

```
public class CrudCaching<TEntity> : ICreateReadUpdate<TEntity>
{
    private TEntity cachedEntity;
    private IEnumerable<TEntity> allCachedEntities;
    private readonly ICreateReadUpdate<TEntity> decorated;

    public CrudCaching(ICreateReadUpdate<TEntity> decorated)
    {
        this.decorated = decorated;
    }

    public void Create(TEntity entity)
    {
        decorated.Create(entity);
    }

    public TEntity ReadOne(Guid identity)
    {
        if(cachedEntity == null)
        {
            cachedEntity = decorated.ReadOne(identity);
        }
        return cachedEntity;
    }
```

8

```
        public IEnumerable<TEntity> ReadAll()
        {
            if (allCachedEntities == null)
            {
                allCachedEntities = decorated.ReadAll();
            }
            return allCachedEntities;
        }

        public void Update(TEntity entity)
        {
            decorated.Update(entity);
        }

    }
```

通过再次应用接口分离原则，你可以把两个用于读取数据的方法组织到它们自己的接口中，然后就可以单独修饰这个只用于读取数据的接口。代码清单8-8展示了新的`IRead`接口，以及相应的缓存修饰器。

代码清单8-8　`ReadCaching`类只用于修饰`IRead`接口

```
    public interface IRead<TEntity>
    {
        TEntity ReadOne(Guid identity);

        IEnumerable<TEntity> ReadAll();
    }
    // . . .
    public class ReadCaching<TEntity> : IRead<TEntity>
    {

        private TEntity cachedEntity;
        private IEnumerable<TEntity> allCachedEntities;

        private readonly IRead<TEntity> decorated;
        public ReadCaching(IRead<TEntity> decorated)
        {
            this.decorated = decorated;
        }

        public TEntity ReadOne(Guid identity)
        {
            if(cachedEntity == null)
            {
                cachedEntity = decorated.ReadOne(identity);
            }
            return cachedEntity;
        }

        public IEnumerable<TEntity> ReadAll()
```

```
        {
            if (allCachedEntities == null)
            {
                allCachedEntities = decorated.ReadAll();
            }
            return allCachedEntities;
        }

    }
```

在你实现最后一个修饰器类前，原始的接口只剩下了两个方法，如代码清单8-9所示。

代码清单8-9　剩余的两个方法也可以统一为单个方法

```
public interface ICreateUpdate<TEntity>
{
    void Create(TEntity entity);

    void Update(TEntity entity);
}
```

除了方法名，Create和Update方法的签名完全相同。除此之外，它们的目的也很相似：前者用于保存一个新建的实体，后者用于保存一个已有的实体。你可以把这二者统一为单个Save方法，该方法内部清楚如何处理新实体和已有实体，而客户端代码不需要知道这些细节。毕竟，客户端很有可能需要同时保存和更新某个实体，如果有了单个可用的接口时就无需两个目的类似的接口了，因为接口的客户端想要做的只是持久化指定的实体。代码清单8-10展示了重构后的接口。

代码清单8-10　ISave接口的实现会创建新的实体，也会适当地更新已经存在的实体

```
public interface ISave<TEntity>
{
    void Save(TEntity entity);
}
```

8

经过这次重构后，你就可以针对该接口添加新的审计修饰器了。每当用户保存一个实体，你都想要增加一些元数据到持久存储中。特别是有关触发保存动作的用户身份和时间信息。代码清单8-11展示的是SaveAuditing修饰器。

代码清单8-11　审计修饰器内部使用了两个ISave接口

```
public class SaveAuditing<TEntity> : ISave<TEntity>
{
    private readonly ISave<TEntity> decorated;
    private readonly ISave<AuditInfo> auditSave;
    public SaveAuditing(ISave<TEntity> decorated, ISave<AuditInfo> auditSave)
    {
        this.decorated = decorated;
        this.auditSave = auditSave;
    }
```

```
        public void Save(TEntity entity)
        {
            decorated.Save(entity);
            var auditInfo = new AuditInfo
            {
                UserName = Thread.CurrentPrincipal.Identity.Name,
                TimeStamp = DateTime.Now
            };
            auditSave.Save(auditInfo);
        }

    }
```

SaveAuditing修饰器本身实现了ISave接口，但也需要两个不同的ISave接口实现来构造修饰器本身。第一个必须是符合修饰器的TEntity泛型类型参数的接口实现，用来完成真正的保存工作（当然，也可以用来进一步修饰完成真正保存工作的实现）。第二个接口实现只针对要保存的AuditInfo类型。虽然上面代码清单中并没有展示AuditInfo类型的定义，但是可以推断它应该包含string类型的UserName属性和DateTime类型的Timestamp属性。当客户端调用Save方法时，会创建一个新的AuditInfo实例并对其属性进行设置。在保存实体数据后，会紧跟着保存该AuditInfo实例包含的新记录数据。

同样，客户端代码应该不清楚这些细节；审计动作对于用户来说是不可察觉的，同时也不会影响到实体数据保存的实现。相似地，ISave<TEntity>接口的叶子实现（也就是用来完成实际保存工作的，没有任何修饰的实现）本身既不知道其他相关修饰器的存在，也不需要为任何具体的修饰做改动。

至此，原来的单个接口已经变为了三个独立的接口，同时，每个接口也都有了自己特有的、有意义的、具有实际功能的修饰器。图8-2展示的UML类图包括了分离原有接口产生的三个新接口和相应的修饰器。

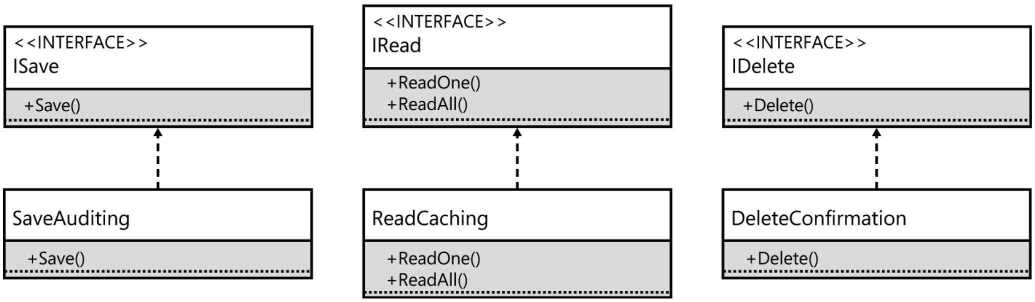


图8-2　接口分离能让你针对必要的方法做修饰，并且不会产生冗余

8.1.3　多重接口修饰

前面章节介绍的所有修饰器和它们修饰的接口都是一对一的关系。这是因为每个修饰器都要

首先实现它要修饰的接口。但是你可以配合应用适配器和修饰器这两个模式来构造多重修饰器，同时也能保证代码量是最少的。

你要创建的下一个修饰器是为了在保存或删除记录时发布一个事件通知。这个通知允许不同的订阅者根据持久存储上的变更做出不同的响应。注意，这个事件通知对于读取数据而言是没有多大用处的，因此这里不会针对`IRead`接口进行修饰。

为了达到这个目的，你首先需要一个发布和订阅事件通知的机制。继续按照接口分离的思路分解，你会得到两个接口，如代码清单8-12所示。

代码清单8-12　分别用于发布和订阅事件通知的两个接口

```
public interface IEventPublisher
{
    void Publish<TEvent>(TEvent @event)
        where TEvent : IEvent;
}
// . . .
public interface IEventSubscriber
{
    void Subscribe<TEvent>(TEvent @event)
        where TEvent : IEvent;
}
```

示例中的`IEvent`接口非常简单，只包含一个`string`类型的`Name`属性。如代码清单8-13所示，使用这两个接口，可以创建在一个实体被删除时发布相应事件通知的修饰器。

代码清单8-13　这个修饰器会在实体被删除时发布一个事件通知

```
public class DeleteEventPublishing<TEntity> : IDelete<TEntity>
{
    private readonly IDelete<TEntity> decorated;
    private readonly IEventPublisher eventPublisher;

    public DeleteEventPublishing(IDelete<TEntity> decorated, IEventPublisher
  eventPublisher)
    {
        this.decorated = decorated;
        this.eventPublisher = eventPublisher;
    }

    public void Delete(TEntity entity)
    {
        decorated.Delete(entity);
        var entityDeleted = new EntityDeletedEvent<TEntity>(entity);
        eventPublisher.Publish(entityDeleted);
    }

}
```

到这一步，你有两个选择：要么在单个修饰器类中同时支持IDelete和ISave两个接口，要么再为ISave接口单独实现一个发布事件通知的修饰器。代码清单8-14展示了第一个选项的实现，它通过增加新的Save方法来让已有的修饰器也支持ISave接口。

代码清单8-14　两个修饰器可以在一个类中实现

```
public class ModificationEventPublishing<TEntity> : IDelete<TEntity>, ISave<TEntity>
{
    private readonly IDelete<TEntity> decoratedDelete;
    private readonly ISave<TEntity> decoratedSave;
    private readonly IEventPublisher eventPublisher;

    public ModificationEventPublishing(IDelete<TEntity> decoratedDelete, ISave<TEntity>
  decoratedSave, IEventPublisher eventPublisher)
    {
        this.decoratedDelete = decoratedDelete;
        this.decoratedSave = decoratedSave;
        this.eventPublisher = eventPublisher;
    }

    public void Delete(TEntity entity)
    {
        decoratedDelete.Delete(entity);
        var entityDeleted = new EntityDeletedEvent<TEntity>(entity);
        eventPublisher.Publish(entityDeleted);
    }

    public void Save(TEntity entity)
    {
        decoratedSave.Save(entity);
        var entitySaved = new EntitySavedEvent<TEntity>(entity);
        eventPublisher.Publish(entitySaved);
    }

}
```

如上面示例所示，只有在多个修饰器共享上下文信息时，在单个类中包含多个修饰器的实现才是有意义的。ModificationEventPublishing修饰器为它实现的ISave和IDelete两个接口实现相同的发布事件通知的功能。但是，将事件发布和审计两种不同目的的修饰器结合在同一个类实现中则是不可取的。因为这样会产生不必要的依赖关系，其中的一个修饰器依赖IEventPublisher接口，而另一个则依赖AuditInfo类型。最好还是把这些不同功能的实现和相应的依赖链分别封装在它们各自的程序集中。

8.2　客户端构建

接口的设计（无论是分离或其他方式产生的）会影响实现接口的类型以及使用该接口的客户端。如果客户端要使用接口，就必须先以某种方式获得接口实例。本章剩余的绝大部分会继续以

手动方式构造接口实现的实例，并把它们通过构造函数参数传递给客户端。另外一种提供接口实例的方式是依赖倒置，下一章会对它进行详细讲解。

为客户端提供接口实例的方式一定程度上取决于接口实现的数目。如果每个接口都有自己特有的实现，那就需要构造所有实现的实例并提供给客户端。或者，如果所有接口的实现都包含在单个类中，那么只需要构建该类的实例就能够满足客户端的所有依赖。

8.2.1 多实现、多实例

继续上面CRUD的示例，假设`IRead`、`ISave`和`IDelete`接口都有自己独有的实现类。接口分离之前，想使用CRUD功能的客户端只需要一个接口，但在接口分离之后，客户端就需要同时引用这三个接口。如代码清单8-15所示。

代码清单8-15 该特定于订单的控制器要求CRUD的每个方面作为一个单独的依赖项

```
public class OrderController
{
    private readonly IRead<Order> reader;
    private readonly ISave<Order> saver;
    private readonly IDelete<Order> deleter;

    public OrderController(IRead<Order> orderReader, ISave<Order> orderSaver,
  IDelete<Order> orderDeleter)
    {
        reader = orderReader;
        saver = orderSaver;
        deleter = orderDeleter;
    }

    public void CreateOrder(Order order)
    {
        saver.Save(order);
    }

    public Order GetSingleOrder(Guid identity)
    {
        return reader.ReadOne(identity);
    }

    public void UpdateOrder(Order order)
    {
        saver.Save(order);
    }

    public void DeleteOrder(Order order)
    {
        deleter.Delete(order);
    }
}
```

示例中的控制器是专门针对订单实体的。这就意味着提供的每个接口都要以Order类作为泛型参数。如果你要在任意一个接口中使用另外一种类型，那么该接口提供的操作也会需要同一种类型。比如，如果你决定将删除接口参数更改为IDelete<Customer>，那么OrderController的DeleteOrder方法就会报错，因为你在尝试用只接受Customers的方法删除一个Order。这就是强类型和泛型在同时起作用。

控制器类中的每个方法都需要一个不同的接口来执行它的功能。为了清晰起见，每个方法都直接调用相应接口的对应方法。当然，实际情况很可能不是这样的。

顾名思义，OrderController只处理Order类。当然，你也可以通过让每个接口参数泛型化来实现一个泛型化的控制器。如代码清单8-16所示。

代码清单8-16 针对实体类型的泛型化控制器类需要每个CRUD接口的泛型参数也都是实体类型

```
public class GenericController<TEntity>
{
    private readonly IRead<TEntity> reader;
    private readonly ISave<TEntity> saver;
    private readonly IDelete<TEntity> deleter;

    public GenericController(IRead<TEntity> entityReader, ISave<TEntity> entitySaver,
  IDelete<TEntity> entityDeleter)
    {
        reader = entityReader;
        saver = entitySaver;
        deleter = entityDeleter;
    }

    public void CreateEntity(TEntity entity)
    {
        saver.Save(entity);
    }

    public TEntity GetSingleEntity(Guid identity)
    {
        return reader.ReadOne(identity);
    }

    public void UpdateEntity(TEntity entity)
    {
        saver.Save(entity);
    }

    public void DeleteEntity(TEntity entity)
    {
        deleter.Delete(entity);
    }
}
```

该示例中的控制器和代码清单8-15中的版本只是略有不同，但是使用二者的客户端代码量却

有着很大的区别。这个泛型化的控制器能够针对任何实体实例化，需要的三个接口也都必须使用相同的实体类型。你再也不能为每个接口提供不同的类型了，比如ISave<Customer>、IRead<Order>、IDelete<LineItem>。

两个版本的控制器都可以按照大致相同的方式创建。在给控制器构造函数传递接口实例前，你必须要先实例化这些接口的相应实现。如代码清单8-17所示。

代码清单8-17 使用依赖的不同实例来创建OrderController

```
static OrderController CreateSeparateServices()
{
    var reader = new Reader<Order>();
    var saver = new Saver<Order>();
    var deleter = new Deleter<Order>();

    return new OrderController(reader, saver, deleter);
}
```

为分离得到的接口分别创建不同的实现类，实际上也就分离了实现。OrderController类的关键点是，它的三个参数（reader、saver和delete）不仅仅是代表三种接口，也代表了三种实现类。

8.2.2 单实现、单实例

另外一种实现多个分离的接口的方式是在单个类中继承并实现它们。第一眼看上去，这种方式也许有些反常（毕竟，接口分离的目的应该不是再次把它们统一在单个实现中吧？），但先不要着急。代码清单8-18展示了同时实现三个接口的单个类。

代码清单8-18 所有接口可以在一个类中实现

```
public class CreateReadUpdateDelete<TEntity> :
      IRead<TEntity>, ISave<TEntity>, IDelete<TEntity>
{
    public TEntity ReadOne(Guid identity)
    {
        return default(TEntity);
    }

    public IEnumerable<TEntity> ReadAll()
    {
        return new List<TEntity>();
    }

    public void Save(TEntity entity)
    {

    }

    public void Delete(TEntity entity)
```

```
    {

    }
}
```

切记，客户端根本不知道有这个类存在。在编译时，客户端代码只知道它所需要的一个个接口。不论底层接口的实现是否还有其他可用的操作，对于客户端而言，每个接口都只包括接口声明的成员。这也是接口封装和隐藏信息的方式。接口就像是实现类上的小窗口，对客户端屏蔽了他们不该看到的部分。

即使单个类同时实现所有接口，上一节基于多实现的控制器示例依然是有效的，因为它的构造函数参数都是对应的接口类。唯一需要改动的就是为要构造的控制器提供参数的方式。如代码清单8-19所示。

代码清单8-19　尽管下面的示例看起来不太正常，但这的确是接口分离已知的副作用之一

```
public OrderController CreateSingleService()
{
    var crud = new CreateReadUpdateDelete<Order>();

    return new OrderController(crud, crud, crud);
}
```

首先，你只需要单个`CreateReadUpdateDelete`类的实例。该类同时实现了三个接口，因此它也能够满足控制器的三个构造函数参数的要求。

看起来不太正常的部分（三个参数是同一个实例）也是合理的，因为每个参数需要的是该类的不同方面。这是接口分离原则产生的一个常见副作用。

与前一节的每个接口都有相应实现的方式相比，这一节的在单个类内实现一系列相关的（但又分离的）接口的方式并不那么通用。后者通常用于接口的叶子实现类，也就是说，既不是修饰器也不是适配器的实现类，而是完成实际工作的实现类。在叶子实现类上应用这种方式，是因为叶子类中所有实现的上下文是一致的。无论你是在使用NHibernate、ADO.NET、Entity Framework或者其他的持久化框架，叶子实现类就是那些直接使用这些库的类。该实例中的所有操作（读取、保存或删除）都是依靠这些库来完成实际工作的。

有些修饰器和适配器也会同时实现一组分离的接口，但是更常见的是为它们提供各个接口的独有实现。

8.2.3　超级接口反模式

代码清单8-20展示了一个常见的错误，那就是由于某些原因又把分离得到的多个接口重新统一在一个聚合接口下。通常这样做是为了避免你在前面几节看到的比较别扭的多个接口类参数的注入。

代码清单8-20　把所有接口分离得来的接口又聚合在同一个接口下是对接口分离原则的错误规避

```
interface IInterfaceSoupAntiPattern<TEntity> : IRead<TEntity>, ISave<TEntity>,
  IDelete<TEntity>
{
}
```

这些接口一起聚合构成了一个“超级接口”，但这显然破坏了接口分离带来的好处。这种超级接口的实现者必然需要再次提供所有操作的实现，而这样做又会把各个修饰器的目标混合在一起。

8.3　接口分离

需要单独修饰接口的能力（或需求）是你想要将大型接口分割为多个小型接口的可能原因之一。尽管如此，我总是把这看作实践中一个足够好的原因。

应用接口分离的另外两个原因分别是客户端和架构的需要。

8.3.1　客户端需要

一般情况下，不同的开发人员会处理不同部分的代码，因此，很可能出现的情况是，当某个开发人员使用另外一个或多个开发人员提供的接口时，他们的工作就会在某一点出现重叠。逐步地详细讲解每个接口不仅不太可能，也是不切实际的。编写任何代码（特别是充分测试的代码）都需要时间，在代码基础上编写扩展的文档，即使是为最终用户，也都会既冗长又费时。相反，更好的办法应该是尽早采用防御方式进行编程，以此阻止其他开发人员（甚至包括将来的自己）无意中使用你的接口做出一些不该做的事情。

切记，客户端只需要它们需要的东西。那些巨型接口倾向于给用户提供更多的控制能力。带有大量成员的接口允许客户端做很多操作，甚至包括它们不应该做的，这样的接口意图很模糊，焦点不明确。所有的类应该总是只有单个清晰的职责。

8

代码清单8-21展示了一个允许客户端和用户设置交互的接口，其中的用户设置是指客户端为应用程序用户界面设置的主题。这个特殊场景下的示例会令你感到惊奇，尽管它是一个只有一个属性的接口，然而依然暴露了太多的东西给客户端。你还能在该接口上做进一步的分离吗？

代码清单8-21　用户配置接口允许访问程序当前的主题

```
public interface IUserSettings
{
    string Theme
    {
        get;
        set;
    }
}
```

首先，来看看代码清单8-22展示的该接口的实现。其中使用了ConfigurationManager类来读写配置文件中的AppSettings数据段。

代码清单8-22　一个从配置文件中加载设置的实现

```
public class UserSettingsConfig : IUserSettings
{
    private const string ThemeSetting = "Theme";

    private readonly Configuration config;

    public UserSettingsConfig()
    {
        config = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
    }

    public string Theme
    {
        get
        {
            return config.AppSettings.Settings[ThemeSetting].Value;
        }
        set
        {
            config.AppSettings.Settings[ThemeSetting].Value = value;
            config.Save();
            ConfigurationManager.RefreshSection("appSettings");
        }
    }
}
```

实现的思路很清楚了，但那又怎样？恰好有两个该接口的客户端，其中一个只想读取数据，而另外一个只想写数据。这正是问题所在，如代码清单8-23所示。

代码清单8-23　接口的不同客户端以不同的目的使用同一个属性

```
public class ReadingController
{
    private readonly IUserSettings settings;

    public ReadingController(IUserSettings settings)
    {
        this.settings = settings;
    }

    public string GetTheme()
    {
        return settings.Theme;
    }
}
// . . .
public class WritingController
```

```
    {
        private readonly IUserSettings settings;

        public WritingController(IUserSettings settings)
        {
            this.settings = settings;
        }

        public void SetTheme(string theme)
        {
            settings.Theme = theme;
        }
    }
```

与期望的一样，ReadingController类只使用了Theme属性的读取器，而Writing-Controller类则只使用了Theme属性的设置器。更糟糕的是，由于接口缺乏分离，你将无法阻止WritingController获取主题数据，也无法阻止ReadingController修改主题数据，后者的问题更大。

为了真正防止和消除错用接口的可能性，你可以将读写分离到各自专有的接口中，如代码清单8-24所示。

代码清单8-24 原有接口被一分为二：一个负责读取主题数据，另外一个负责修改

```
public interface IUserSettingsReader
{
    string Theme
    {
        get;
    }
}

// . . .
public interface IUserSettingsWriter
{
    string Theme
    {
        set;
    }
}
```

虽然看起来有些奇怪，但这绝对是有效的C#代码。也许接口只提供属性的读取器还很平常，但是属性值提供设置器的情况就很少见了。

两个控制器现在可以只依赖它们实际需要的接口。如代码清单8-25所示，Reading-Controller类只实现了IUserSettingReader接口，而WritingController类只实现了IUserSettingWriter接口。

8

代码清单8-25　这两个接口现在分别只依赖它们真正需要的接口

```
public class ReadingController
{
    private readonly IUserSettingsReader settings;

    public ReadingController(IUserSettingsReader settings)
    {
        this.settings = settings;
    }

    public string GetTheme()
    {
        return settings.Theme;
    }
}
// . . .
public class WritingController
{
    private readonly IUserSettingsWriter settings;

    public WritingController(IUserSettingsWriter settings)
    {
        this.settings = settings;
    }

    public void SetTheme(string theme)
    {
        settings.Theme = theme;
    }
}
```

通过接口分离,你已经防止了`ReadingController`类对用户设置的修改,也防止了`WritingController`类对用户设置的读取。因此，开发人员也不会无意地错用控制器来完成它们不应该支持的操作。

如代码清单8-26所示，依然使用`ConfigurationManager`类的两个接口的实现类，只需要些许改动即可。

代码清单8-26　`UsersSettingsConfig`类现在同时实现了两个接口，但是使用这些接口的客户端对此并不知情

```
public class UserSettingsConfig : IUserSettingsReader, IUserSettingsWriter
{
    private const string ThemeSetting = "Theme";

    private readonly Configuration config;

    public UserSettingsConfig()
    {
        config = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
    }
```

```
        public string Theme
        {
            get
            {
                return config.AppSettings.Settings[ThemeSetting].Value;
            }
            set
            {
                config.AppSettings.Settings[ThemeSetting].Value = value;
                config.Save();
                ConfigurationManager.RefreshSection("appSettings");
            }
        }
    }
```

上面示例中的实现类，除了同时继承读写两个接口外，与前面代码清单8-22中的该类完全一致。记住，这个相同的实现既能传递给ReadingController类，又能传递给WriterController类，然而，这两个类上的接口所形成的窗口会让相应的获取和设置操作变得不可用。

有些客户端能读但不能写的需求在实际中很常见，但是另外一个能写却不能读的场景是很少见的。要处理这种情况，你可以先分离读接口，然后让写接口继承读接口，而不是采取上面介绍的完全分离。如代码清单8-27所示。

代码清单8-27 通过使用方法代替属性，现在的IUserSettingWriter接口可以继承IUser-SettingReader接口

```
    public interface IUserSettingsReader
    {
        string GetTheme();
    }
    // . . .
    public interface IUserSettingsWriter : IUserSettingsReader
    {
        void SetTheme(string theme);
    }
```

为了做到这一点，需要将Theme属性转化为GetTheme和SetTheme这两个方法。这是因为C#还不能很好地支持属性继承。在这个示例中，C#要求Theme属性在读和写接口中都要有。尽管类能够很好地实现来自两个不同接口的一个接口的get和set部分，但不幸的是，使用接口继承时，情况并非如此。当接口继承中出现属性命名冲突时，编译器会给出基类属性会被子类属性覆盖隐藏的警告。这样就无法达成你想要的目的。也不要使用编译器给出的在基类中使用new关键字标记属性的建议，因为那样同样无法继承基类同名属性的读取器。

此时，你可以用方法实现属性同样的语法功能。GetTheme方法和Theme属性的读取器一样，SetTheme方法则与Theme属性的设置器一样。现在继承的结果会与你期望的一样：读接口的实施者和客户端都只能访问GetTheme方法，而写接口的实施者和客户端不仅能访问SetTheme方法，

也能访问GetTheme方法。此外，任何IUserSettingsWriter接口的实现都自动是一个IUserSettingsReader接口的实现。

代码清单8-28展示了对WritingController的一处改动：在设置新主题前检查主题是否被改动过。现在这个场景是合理的，因为用户设置的改写服务同时也是用户设置的读取服务。这种情况下，就无需单独分别调用读和写这两个接口。

代码清单8-28　WritingController类通过一个接口就能同时访问读取器和设置器

```
public class WritingController
{
    private readonly IUserSettingsWriter settings;

    public WritingController(IUserSettingsWriter settings)
    {
        this.settings = settings;
    }

    public void SetTheme(string theme)
    {
        if (settings.GetTheme() != theme)
        {
            settings.SetTheme(theme);
        }
    }
}
```

授权

另外一个按照客户需求来做接口分离的示例是，应用程序在某个特定状态下只能提供一组特定的操作。比如，根据登录状态，用户可以执行的操作通常会是不同的。

代码清单8-29展示的未授权接口只包括匿名且未授权用户可以执行的操作。

代码清单8-29　该接口只包含匿名用户可以执行的操作

```
public interface IUnauthorized
{
    IAuthorized Login(string username, string password);

    void RequestPasswordReminder(string emailAddress);
}
```

注意到，Login方法的返回值是另外一个接口类。返回该接口的有效实例的前提是用户提供的凭据必须完全正确，之后客户端就可以执行已授权的操作了，如代码清单8-30所示。

代码清单8-30　登录后，用户将可以访问已授权的操作

```
public interface IAuthorized
{
    void ChangePassword(string oldPassword, string newPassword);
```

```
        void AddToBasket(Guid itemID);

        void Checkout();

        void Logout();
    }
```

用户只有在输入凭据且成功经过授权登录后，才可以使用IAuthorized接口提供的所有操作。

按照客户端需求进行接口分离能够阻止开发人员意外地做出他们不应该做的事情。在上面的示例中，它能够阻止匿名用户执行已授权的操作。当然，总会有些意外情况，但是通过这种设计可以让开发人员认识到，如果他们想要做一些他们不应该做的事情，就需要改动应用程序的核心基础。

8.3.2　架构需要

另外一种接口分离的驱动力来自于架构设计。高层设计产生的决定对底层代码的组织有着非常大的影响。

下面的示例中，高层设计的决定是采用非对称架构。与前面几节展示的读写分离类似，代码清单8-31展示的IPersistence接口包含了一些查询和命令的组合。

代码清单8-31　IPersistence接口既包括命令，也包括查询

```
    public interface IPersistence
    {
        IEnumerable<Item> GetAll();

        Item GetByID(Guid identity);

        IEnumerable<Item> FindByCriteria(string criteria);

        void Save(Item item);

        void Delete(Item item);
    }
```

该接口的非对称架构就是命令查询责任分离模式的一部分。这里再次出现分离（segregation）这个词并不是巧合，因为这个架构模式本身的意图就是指导你去做一些接口分离的动作。

代码清单8-32展示了IPersistence接口的第一种实现。

代码清单8-32　当命令和查询的处理非对称时，实现一片混乱

```
    public class Persistence : IPersistence
    {
        private readonly ISession session;
        private readonly MongoDatabase mongo;

        public Persistence(ISession session, MongoDatabase mongo)
```

8

```
        {
            this.session = session;
            this.mongo = mongo;
        }

        public IEnumerable<Item> GetAll()
        {
            return mongo.GetCollection<Item>("items").FindAll();
        }

        public Item GetByID(Guid identity)
        {
            return mongo.GetCollection<Item>("items").FindOneById(identity.ToBson());
        }

        public IEnumerable<Item> FindByCriteria(string criteria)
        {
            var query = BsonSerializer.Deserialize<QueryDocument>(criteria);
            return mongo.GetCollection<Item>("Items").Find(query);
        }

        public void Save(Item item)
        {
            using(var transaction = session.BeginTransaction())
            {
                session.Save(item);

                transaction.Commit();
            }
        }

        public void Delete(Item item)
        {
            using(var transaction = session.BeginTransaction())
            {
                session.Delete(item);

                transaction.Commit();
            }
        }
    }
```

示例中有两个截然不同的依赖：NHibernate用于执行命令，MongoDB用于执行查询。前者是一个配合域模型使用的对象关系映射器，后者是一个用于快速查询的文档存储库。现在这个类型有了两个完全无关的依赖，因此也就有了两处改变点。因为依赖的不同，所以依赖相关的修饰器也很有可能不同。这里不会像前面的CRUD示例一样将整个接口分割为多个独立的小操作，而只是把接口一分为二：命令和查询。图8-3中的UML类图展示了重构后的结构。

将命令和查询分离到各自的接口后，它们的实现就能够完全依赖各自必需的包。命令接口的实现将只依赖NHibernate包，而查询接口的实现则只依赖MongoDB包。

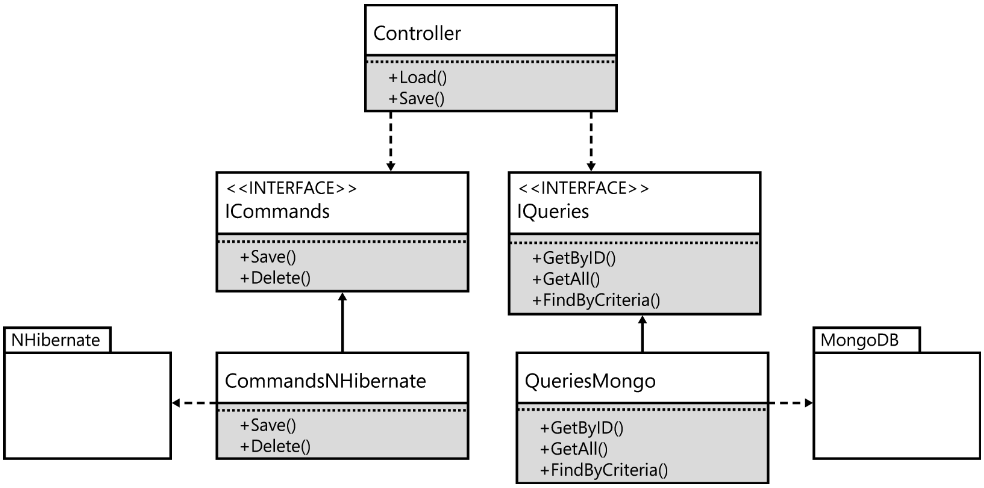


图8-3　根据架构需要分离接口能让各个实现带有自己必需的依赖

理想情况下，这两个接口的实现不仅仅是不同的类，而且也会依赖不同的包（程序集）。如果依然有着相同的依赖，仅仅通过接口分离只能解决部分问题。因为它们的实现和各自的依赖链是相互关联的，所以无法单独重用其中的某个实现。

代码清单8-33展示了分离后的两个接口。现在可以分别实现它们了。

代码清单8-33　接口分离为了查询和命令方法

```
public interface IPersistenceQueries
{
    IEnumerable<Item> GetAll();

    Item GetByID(Guid identity);

    IEnumerable<Item> FindByCriteria(string criteria);
}
// . . .
public interface IPersistenceCommands
{
    void Save(Item item);

    void Delete(Item item);
}
```

查询接口的类实现和前面的`Persistence`类中的查询部分实现完全一样，只是不再包括任何命令相关的代码，也就是说，没有任何对NHibernate包的依赖，如代码清单8-34所示。

代码清单8-34　查询实现仅依赖MongoDB

```
public class PersistenceQueries : IPersistenceQueries
{
    private readonly MongoDatabase mongo;
```

```
    public Persistence(MongoDatabase mongo)
    {
        this.mongo = mongo;
    }

    public IEnumerable<Item> GetAll()
    {
        return mongo.GetCollection<Item>("items").FindAll();
    }

    public Item GetByID(Guid identity)
    {
        return mongo.GetCollection<Item>("items").FindOneById(identity.ToBson());
    }

    public IEnumerable<Item> FindByCriteria(string criteria)
    {
        var query = BsonSerializer.Deserialize<QueryDocument>(criteria);
        return mongo.GetCollection<Item>("Items").Find(query);
    }
}
```

同样，命令接口的类实现也不再需要任何对MongoDB包的引用，如代码清单8-35所示。

代码清单8-35　命令实现仅依赖NHibernate

```
public class PersistenceCommands : IPersistenceCommands
{
    private readonly ISession session;
    public PersistenceCommands(ISession session)
    {
        this.session = session;
    }

    public void Save(Item item)
    {
        using(var transaction = session.BeginTransaction())
        {

            session.Save(item);

            transaction.Commit();
        }
    }

    public void Delete(Item item)
    {
        using(var transaction = session.BeginTransaction())
        {
            session.Delete(item);

            transaction.Commit();
```

```
        }
    }
}
```

8.3.3 单方法接口

接口分离会生成很小的接口。接口规模越小，就变得越通用。.NET Framework中有很多作用类似的委托，比如：Action、Func和Predicate等。但是，委托并不像接口那样通用。尽管委托也有自己的用途，但是以各种方式分离产生的接口不仅可以被修饰和适配，还可以再次组合。因为接口必须实现，在同样的实现类中也可以通过其他接口或构造函数参数获得更多的上下文信息。

最简单的接口只有单个方法。最简单的方法既没有参数也没有返回值。如代码清单8-36所示。

代码清单8-36 ITask是一个最简单的接口

```
public interface ITask
{
    void Do();
}
```

这个接口非常适合修饰。因为它没有返回值，所以甚至可以引用异步的即发即弃修饰器。客户端在需要发送消息时，如果无需提供任何上下文信息并且无需等待任何响应，就可以使用这个接口。

在此基础上改进可以得到一个IAction接口，它和.NET Framework中的Action委托有些类似，需要一个泛型参数来表示上下文信息。如代码清单8-37所示。

代码清单8-37 IAction接口中增加了一个上下文参数

```
public interface IAction<TContext>
{
    void Do(TContext context);
}
```

IAction只是比ITask复杂一点点。如果你想有一个返回值而不是参数，你就创建了一个IFunction接口，如代码清单8-38所示。

代码清单8-38 IFunction接口有返回值

```
public interface IFunction<TReturn>
{
    TReturn Do();
}
```

在IFunction接口基础上，如果你需要返回一个布尔值。那么你就得到了一个IPredicate

接口，如代码清单8-39所示。

代码清单8-39　IPredicate是一个返回布尔值的函数

```
public interface IPredicate
{
    bool Test();
}
```

IPredicate接口可以用于封装一个分支测试，比如一个if语句或者循环中的判断从句。

尽管这些接口看起来很不起眼，但是通过修饰、适配和组合大量不同种类的这种单方法接口，你可以完成很多复杂的事情。

8.4　总结

本章旨在介绍如何设计优秀的接口。很多时候，接口只是隐藏在它们后面的大规模子系统的外观。在某些临界点，接口会失去它们的适应能力，而这些适应能力正是让接口成为开发SOLID代码基础的关键因素。

应该分离接口的原因有很多，比如用来辅助修饰，为客户端隐藏他们不该看到的功能，为其他开发人员提供自文档，以及作为架构设计的产物等。无论是哪种原因，你都应该在创建任何接口时记得接口分离这个技术原则。与大多数编程工具一样，最好是从开始阶段就应用接口分离原则，而不是到了后期才去辛苦地重构。

第 9 章

依赖注入原则

完成本章学习之后，你将学到以下技能。

- 理解依赖注入的重要性。
- 用依赖注入将SOLID代码组织在一起。
- 在穷人的依赖注入、控制反转容器或约定优于配置这三种方式中选择其一。
- 避免依赖注入反模式。
- 围绕组合根和解析根来组织你的解决方案。
- 知道如何正确地结合工厂模式和依赖注入来管理对象的生命周期。

依赖注入（Dependency Injection，DI）是一个非常简单的概念，实现起来也很简单。尽管如此，这种简单性却掩盖了该模式的重要性。没有依赖注入，前面几章介绍的SOLID技术（以及前面与敏捷基础相关的章节）都不可能实际应用。

当某些事情很简单也很重要时，人们就会将它过度复杂化。依赖注入也一样，但是在应用它时，的确有几个陷阱要留意。这些陷阱包括混淆该模式意图的各种反模式和不良习惯。

正确实现的依赖注入对于项目的绝大多数代码而言是不可见的。它们被局限在一个很小的代码范围内，通常是在一个独立的程序集内。最好是从一开始就应用依赖注入，因为在已经建立的项目中引入依赖注入会既困难又耗时。

9.1 简单的开始

下面的示例会突出展示依赖注入能够解决的潜在问题。假设你在开发一个用户可以用来管理待办事项清单的任务管理应用程序。此外，还假定此时项目依然处于早期的开发阶段，同时也决定了使用WPF开发用户界面。此时，你已经有了一个只能从持久存储中读取并显示待办事项列表的主窗口。如图9-1所示。

因为是一个WPF应用程序，你正在使用模型-视图-视图模型（Model-View-ViewModel，MVVM）模式确保隔离各个层之间的依赖。虽然还没有使用依赖注入，但是该应用程序已经在努力使用其他的最佳实践。其中的一个视图模型是主窗口的后台控制器。`TaskListController`类实例委托一个`TaskService`类实例来获取所有代办事项数据。代码清单9-1展示了一个在没有应用依赖注入的情况下达到设计目的的实例。

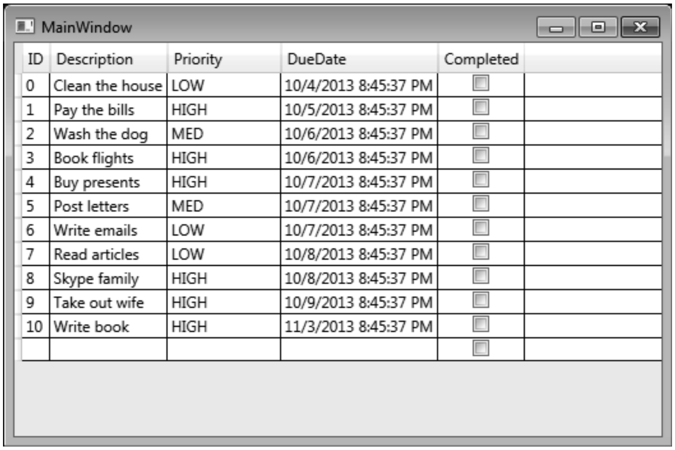


图9-1　除了描述外，待办事项还包括了优先级、截止日期和完成情况等状态

代码清单9-1　控制器并没有使用依赖注入

```
{
    public event PropertyChangedEventHandler PropertyChanged = delegate { };

    private readonly ITaskService taskService;
    private readonly IObjectMapper mapper;
    private ObservableCollection<TaskViewModel> allTasks;

    public TaskListController()
    {
        this.taskService = new TaskServiceAdo();
        this.mapper = new MapperAutoMapper();

        var taskDtos = taskService.GetAllTasks();
        AllTasks = new
  ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
    }

    public ObservableCollection<TaskViewModel> AllTasks
    {
        get
        {
            return allTasks;
        }
        set
        {
            allTasks = value;
            PropertyChanged(this, new PropertyChangedEventArgs("AllTasks"));
        }
    }

}
```

上面示例中的实现方式存在如下一些问题。

- 很难做单元测试，因为控制器依赖某个具体实现。
- 不清楚视图模型的依赖点，除非查看它的源代码。
- 隐式地依赖服务的所有依赖。
- 无法灵活地替换服务实现。

本节的剩余内容会通过对比原始类和应用依赖注入的重构版本来逐个深入剖析这些问题。代码清单9-2展示了应用了依赖注入的重构版本，其中的改动已经加粗。

代码清单9-2 重构后，控制器使用了依赖注入

```
public class TaskListController : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged = delegate { };

    private readonly ITaskService taskService;
    private readonly IObjectMapper mapper;
    private ObservableCollection<TaskViewModel> allTasks;

    public TaskListController(ITaskService taskService, IObjectMapper mapper)
    {
        this.taskService = taskService;
        this.mapper = mapper;
    }

    public void OnLoad()
    {
        var taskDtos = taskService.GetAllTasks();
        AllTasks = new
  ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
    }

    public ObservableCollection<TaskViewModel> AllTasks
    {
        get
        {
            return allTasks;
        }
        set
        {
            allTasks = value;
            PropertyChanged(this, new PropertyChangedEventArgs("AllTasks"));
        }
    }

}
```

要单独测试代码清单9-1中的`TaskListController`类的第一个版本，就需要模拟`Task-Service`类。然而，很难通过常规方式去模拟`TaskService`类，而`TaskService`类并不是个可代理的类，或者说你应该把它改造成可代理的。代码清单9-2中是`TaskListController`类的第二个

版本，它仅接受ITaskService接口，而不是像第一个版本中那样直接依赖某个实现类。这样重构后就更容易测试了，因为接口实例总是可以被替换的。

> **注意**　只有能为用户提供替换实现（也称为**代理**，proxy）的类才称之为是可代理的。可代理的类必须把所有方法声明为虚方法，而接口总是直接可代理的。

如果一个类在它的方法内部能随意构造类的实例，你就无法从外部知道该类到底需要什么才可以正常工作。没有应用依赖注入的第一个示例就是一个依赖的黑盒子。你只有通过查看类实现的源代码才能知道真相，因为它没有通过该类的接口或者方法签名声明任何依赖。第二个示例中应用了依赖注入，它清楚地表明了需要一个任务服务的实现才能正常工作。客户端代码可以通过Visual Studio的智能感知特性看到构造函数的签名。

当类A和类B之间存在依赖关系时，如果类B依赖类C，那么类A也隐式地依赖类C。随从反模式就是这样引入了交错复杂的依赖关系网，而这种依赖关系网一旦形成，就很难再整理清楚。如果你能确保你的接口对自己行为做了正确的抽象，那么客户在使用该接口时就不再需要其他任何东西了。即使该接口的实现可能依赖了一些大型的外部组件，比如数据库，也不会影响到使用该接口的客户端代码。这就是正确应用阶梯模式的结果。

直接实例化实现对象，你也会失去接口能提供的另外一个扩展能力：你将无法继承TaskService类并增强已有方法的功能。即使方法已经被声明为虚方法也一样，因为无论如何你都要改动客户端代码去直接实例化这个派生的子类。接口允许应用各种强大的模式来为自己提供多种实现或增强现有的实现。此外你也已经知道，即使接口的初始版本类实现已经编写好，只要新的接口需求还没出现，现有接口的这种允许增加新的实现或增强现有实现的扩展能力是一直存在的。这也是代码适应能力的关键点。

9.1.1　任务列表应用

图9-2展示了你使用任务列表应用想要实现的包级别和类级别组织。

用户界面层包括了WPF、控制器以及视图模型相关的代码。服务层通过一个接口对控制器的依赖进行了抽象，它的实现简单地使用了ADO.NET来从持久存储中获取所有任务数据。

服务实现返回的TaskDto类是从存储中获取的一行任务数据在内存中的表示。这只是一个普通的CLR对象（Plain Old CLR Object，POCO），就其本身而言，它并不具备WPF视图模型应有的丰富功能。因此，当TaskController类从ITaskService接口获取TaskDto类的对象时，它会请求一个IObjectMapper接口来把这些TaskDto类的对象转换为TaskViewModel类的对象，后者实现了INotifyPropertyChanged接口，所以可以与其他的视图相关的特性结合使用。

代码清单9-3展示了ITaskService接口的ADO.NET实现。你主要会担心的是类的构造函数，此外，本章后面还会讨论这个类中另一个很难消除的代码味道。

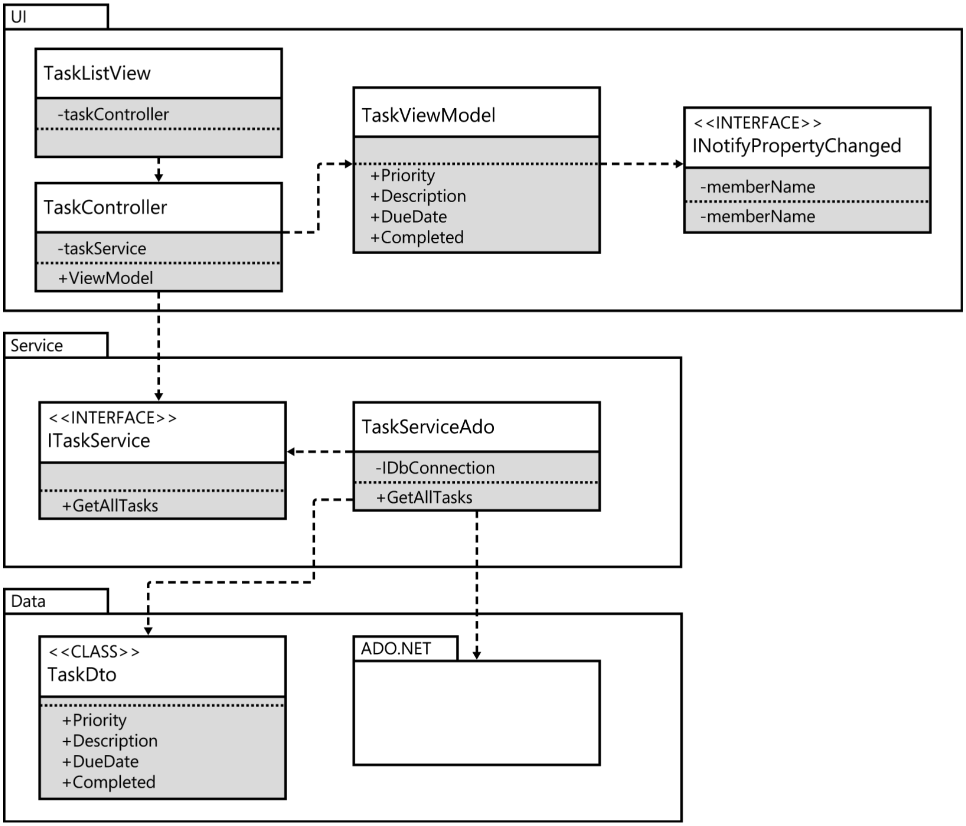


图9-2　分三层的任务列表应用的UML类图，包含包

代码清单9-3　TaskService负责检索任务列表数据

```
public class TaskServiceAdo : ITaskService
{
    public TaskServiceAdo(ISettings settings)
    {
        this.settings = settings;
    }

    public IEnumerable<TaskDto> GetAllTasks()
    {
        var allTasks = new List<TaskDto>();


    private readonly ISettings settings;

    private const int IDIndex = 0;
    private const int DescriptionIndex = 1;
    private const int PriorityIndex = 2;
```

```
    private const int DueDateIndex = 3;
    private const int CompletedIndex = 4;
        using(var connection = new
  SqlConnection(settings.GetSetting("TaskDatabaseConnectionString")))
        {
            connection.Open();

            using(var transaction = connection.BeginTransaction())
            {
                var command = connection.CreateCommand();
                command.Transaction = transaction;
                command.CommandType = CommandType.StoredProcedure;
                command.CommandText = "[dbo].[get_all_tasks]";

                using(var reader = command.ExecuteReader(CommandBehavior.CloseConnection))
                {
                    if (reader.HasRows)
                    {
                        while (reader.Read())
                        {
                            allTasks.Add(
                                new TaskDto
                                {
                                    ID = reader.GetInt32(IDIndex),
                                    Description = reader.GetString(DescriptionIndex),
                                    Priority = reader.GetString(PriorityIndex),
                                    DueDate = reader.GetDateTime(DueDateIndex),
                                    Completed = reader.GetBoolean(CompletedIndex)
                                }
                            );
                        }
                    }
                }
            }
        }

        return allTasks;
    }

}
```

ISettings接口从TaskService类中抽象了获取连接字符串的细节。该接口的一个可能实现就是直接适配Microsoft .NET Framework提供的ConfigurationManager类。不难想象代码中某处肯定会使用ISettings接口来存储设置数据。另外一个问题就是ConfigurationManager类是静态的，因此难以进行模拟。直接使用该静态类会给获取诸如连接字符串等应用程序配置带来局限，也会降低TaskServiceAdo类的可测性。

9.1.2　对象图的构建

本书前面已经多次提到，接口实例要注入到构造函数中。当然，只注入接口实例是不够的，

你还必须要提供一个接口的实现。有两种主要的注入方式：穷人的依赖注入和控制反转容器。为了展示依赖注入的工作原理，本节会先讲解穷人的依赖注入。

1. 穷人的依赖注入

顾名思义，穷人的依赖注入（Poor Man's Dependency Injection），这个模式不需要任何其他外部依赖就可以实现注入。它需要提前为控制器创建必需的对象图。代码清单9-4展示了如何构建重构后的TaskListController类以及如何给它传递作为应用程序主窗口的TaskListView类实例。

代码清单9-4　穷人的依赖注入比较繁琐但却很灵活

```
public partial class App : Application
{
    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        var settings = new ApplicationSettings();
        var taskService = new TaskServiceAdo(settings);
        var objectMapper = new MapperAutoMapper();
        controller = new TaskListController(taskService, objectMapper);
        MainWindow = new TaskListView(controller);
        MainWindow.Show();

        controller.OnLoad();
    }

    private void CreateMappings()
    {
        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();
    }

    private TaskListController controller;
}
```

示例代码是你的应用程序入口。OnApplicationStartup方法是一个WPF内部事件处理器，用它来完成一些事情的初始化。尽管不同类型的应用程序的入口代码会有所不同，但是它始终是一个放置依赖注入代码的好地方。

应用程序入口的引导过程很简单。目标是创建一个TaskListView类实例，因为这个视图类是整个应用程序解决方案的解析根。解析根会在本章后面做详细介绍。为了创建TaskListView类，你首先要需要一个TaskListController实例。而后者又需要一个ITaskService接口实例和一个IObjectMapper接口实例，所以，你又得先初始化这两个接口的实例，此时你就需要提供这个接口的实现了。而ITaskService接口的实现类TaskServiceAdo又需要一个ISettings接口的实现，所以你需要先提供一个ApplicationSettings类的实例。ApplicationSettings类本身则是.NET Framework中的ConfigurationManager类的一个适配器。图9-3中的UML类图说明了这些依赖。

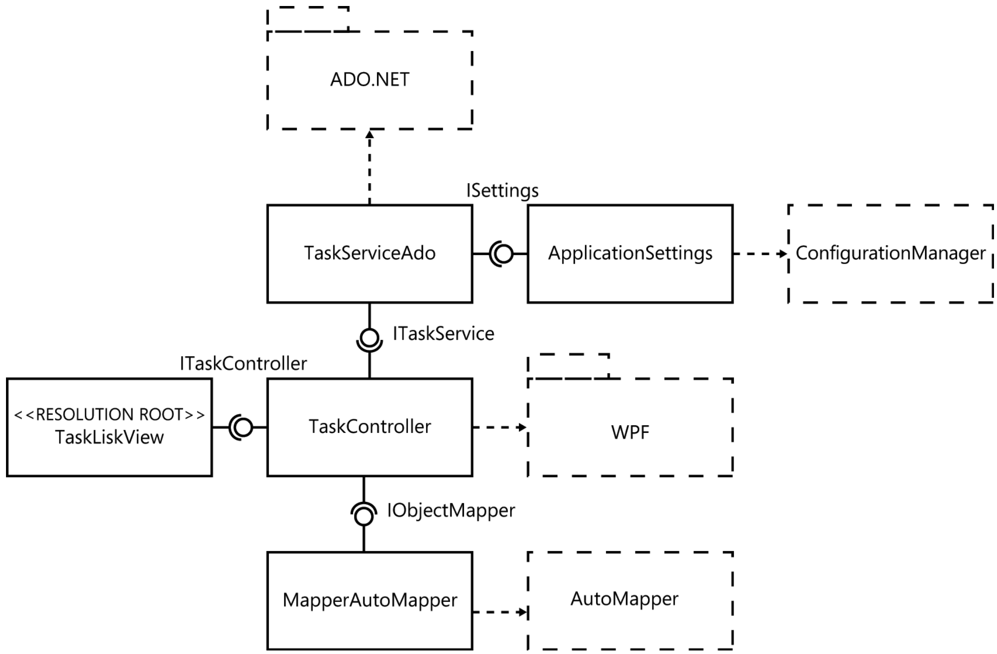


图9-3　任务列表应用程序由一组接口、接口的实现以及相关的依赖构成

每个类都依赖一个或多个可能也需要依赖的其他类。诸如`MapperAutoMapper`类和`ApplicationSettings`类之类的适配器实现很常见，它们通常只满足接口需要的依赖，但实际上依然是委托其他类完成实际工作。即使不是适配器的类，也会委托自己的一些依赖来做一些工作，比如`TaskServcieAdo`类，它实际上是使用ADO.NET直接获取数据。`ITaskService`接口的其他实现可以从其他地方获取数据，比如，可以实现一个主要委托NHibernate完成实际动作的`TaskServiceNHibernate`类。此外，也可以实现一个依赖Microsoft Outlook插件框架的从Outlook内置的任务清单中读取任务数据的`TaskServcieOutlook`类。再强调一遍，只要符合接口的任何东西都可以是任务的数据源，因为接口本身从不会把自己和任何具体的实现技术绑定在一起。

穷人的依赖注入会比较冗长。当该应用程序要扩展支持增加新任务、编辑任务以及可能的任务提醒功能时，你会很容易预见，为依赖注入构造各种实例的代码会快速增长，慢慢地，它们就会变得不是那么容易理解了。尽管如此，穷人的依赖注入方式仍然很灵活。无论你要构造多么复杂的对象图，构造的方式都是清清楚楚的。因为方式只有一种：手动创建任何需要的实例，然后把它们传递给聚合它们功能的类，重复这个动作直到成功实例化应用程序的解析根。在这里，你可以为要实例化的类所依赖的接口应用任何意图的修饰器，也就是说，穷人的依赖注入允许你去随意定制要构建的对象图。

2. 方法注入

不只是构造函数可以为类注入依赖项。方法和属性成员也都可以，只是它们和构造函数的使用场景有所不同。

代码清单9-5展示了再次重构的TaskListController类，ITaskService接口的GetAllTasks方法参数可以接受ISettings接口实例的注入。这需要改动ITaskService接口的方法签名。

代码清单9-5 现在，任务服务实例是从方法的参数中获取设置实例，而不是从构造函数的参数中获取

```
public class TaskListController : INotifyPropertyChanged
{
    public TaskListController(ITaskService taskService, IObjectMapper mapper, ISettings
  settings)
    {
        this.taskService = taskService;
        this.mapper = mapper;
        this.settings = settings;
    }

    public void OnLoad()
    {
        var taskDtos = taskService.GetAllTasks(settings);
        AllTasks = new
  ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
    }
}
```

如果只有该方法需要这个依赖时，从方法参数注入依赖会很有用。从构造函数注入依赖表明类中的多数行为需要该依赖项，但是如果只有少部分方法需要某个依赖，从各个方法参数注入该依赖会更好。方法注入方式也有缺点，那就是，用户在调用方法前必须要先准备好依赖的实例。客户端可以通过构造函数或者方法参数沿着调用栈一直将依赖实例传递给需要使用该依赖的目标类。

3. 属性注入

与方法注入类似，属性也可以用于注入依赖。代码清单9-6展示了再次重构的TaskList-Controller类。这里的ITaskService接口改用属性Settings来注入依赖。再说一次，要切记，需要同时改动接口和实现来支持相应的属性。

代码清单9-6 也可以通过属性来完成依赖注入的动作

```
public class TaskListController : INotifyPropertyChanged
{
    public TaskListController(ITaskService taskService, IObjectMapper mapper, ISettings
  settings)
    {
        this.taskService = taskService;
        this.mapper = mapper;
        this.settings = settings;
    }

    public void OnLoad()
    {
```

```
            taskService.Settings = settings;
            var taskDtos = taskService.GetAllTasks();
            AllTasks = new
    ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
        }
    }
```

这种方式的好处是可以在运行时改变属性实例值。从构造函数注入的依赖实例在类的整个生命周期内都可以使用，而从属性注入的依赖实例还能从类生命周期的某个中间点开始起作用。

9.1.3　控制反转

本书通篇的示例都展示这样的场景：开发中的类委托某些抽象完成动作，而这些被委托的抽象又被其他的类实现，这些类又会去委托其他一些抽象完成某些动作。最终，在依赖链终结的地方，都是一些小且直接的类，它们已经不需要任何依赖了。要构造有依赖项的类，首先要构造并注入这些依赖项。你已经知道如何通过手动构造类实例并把它们传递给构造函数的方式来实现依赖注入的效果。尽管这种方式已经可以让你任意替换或修饰依赖的实现，但是构造的实例对象图依然是静态的，也就是说，编译时就已经确定了的。控制反转（Inversion of Control，IoC）允许你将构建对象图的动作推迟到运行时。

控制反转的概念通常都是在控制反转容器（container）的上下文中出现。控制反转容器组成的系统能够将应用程序使用的接口和它的实现类关联起来，并能在获取实例的同时解析所有相关的依赖。

代码清单9-7中展示的应用程序入口代码中使用了Unity控制反转容器。代码的第一步就是初始化得到一个`UnityContainer`实例。注意，示例代码中这样实例化控制反转容器是在直接实例化基础组件，在后期想替换为其他容器会比较困难。

代码清单9-7　示例中没有使用手动构造实现的实例，而是通过使用控制反转容器来建立类和接口的映射关系

```
public partial class App : Application
{
    private IUnityContainer container;
    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        container = new UnityContainer();
        container.RegisterType<ISettings, ApplicationSettings>();
        container.RegisterType<IObjectMapper, MapperAutoMapper>();
        container.RegisterType<ITaskService, TaskServiceAdo>();
        container.RegisterType<TaskListController>();
        container.RegisterType<TaskListView>();

        MainWindow = container.Resolve<TaskListView>();
        MainWindow.Show();
```

```
        ((TaskListController)MainWindow.DataContext).OnLoad();
    }

    private void CreateMappings()
    {
        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();
    }

}
```

在创建好Unity容器后，你需要告诉该容器应用程序生命周期内每个接口对应的具体实现类是什么。Unity遇到任何接口时，它都会知道需要去解析哪个实现。如果你没有为某个接口指定对应的实现类，Unity会提醒你该接口无法实例化。

在完成接口和对应实现类的关系注册后，你需要获得一个应用程序的解析跟，也就是TaskListView类的实例。Unity容器的Resolve方法会检查TaskListView类的构造函数，然后尝试去实例化构造函数要注入的依赖项，如此反复，直到完全实例化整个依赖链上的所有依赖项的实例后，Resolve方法会成功实例化TaskListView类的实例。这和穷人的依赖注入方式的手动构造过程完全一样，不同的只是后者需要你去手动检查构造函数并直接实例化你看到的依赖项类。

1. 注册、解析、释放模式

所有的控制反转容器都符合一个只有三个方法的简单接口，如代码清单9-8所示。Unity也不例外，它遵守一个类似的模式。

代码清单9-8　尽管每个控制反转容器的实现不完全相同，但是都符合下面这个通用的接口

```
public interface IContainer : IDisposable
{
    void Register<TInterface, TImplementation>()
        where TImplementation : TInterface;

    TImplementation Resolve<TInterface>();

    void Release();
}
```

9

这三个方法的目的解释如下所示。

- Register：应用程序初始化会首先调用该方法。而且该方法会被多次调用以注册很多不同的接口及其实现之间的映射关系。这里的where子句用来强制TImplementation类型必须实现它所继承的TInterface接口。该方法还支持注册某个已经构造好的实例和一个没有指定接口的类型的映射关系，这样做，可以注册该类型和这个实例所实现的所有接口之间的映射关系。
- Resolve：应用程序运行时会调用该方法。通常一组特殊的类会被自动解析为对象图中的顶层对象。比如，使用模型–视图–控制器（MVC）模式的ASP.NET应用程序中的控制器对象，使用视图模型优先模式的WPF应用程序中的视图模型对象，以及使用模型–视图–

表示器（MVP）的Windows Form应用程序中的视图对象。也就是说，你不应该在你的应用程序类中对这些顶层对象（控制器、视图、表示器、服务、域、业务逻辑或数据访问等）调用Resolve方法。

- Release：应用程序生命周期中，这些类的实例不再需要时，就可以释放它们占有的资源了。这很有可能发生在应用程序结束时，但也有可能发生在应用程序运行时的某些恰当时机。比如，在网络应用场景中，通常情况下，资源只对单次请求（per-request）有效。因此，每次请求后都会调用Release方法。有关对象生命周期的问题会在后面更详细地讨论。
- Dispose：如上面示例代码中所示，大多数控制反转容器也都会实现IDisposable接口。应用程序在关闭的时候会调用该方法。Dispose方法和Release方法不一样，它会清除容器内部的字典，这样它不再带有映射关系的注册信息，所以也就无法解析任何依赖了。

可以对清单9-7展示的第一个控制反转容器示例进行重构，重构后，所有对容器的调用都被封装在一个类内。这样做就可以把冗长的注册代码从WPF应用程序的后置代码中隔离开。如代码清单9-9所示。

代码清单9-9　启动事件处理器委托配置类来完成容器相关的工作

```
public partial class App : Application
{
    private IocConfiguration ioc;

    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        ioc = new IocConfiguration();
        ioc.Register();

        MainWindow = ioc.Resolve();
        MainWindow.Show();

        ((TaskListController)MainWindow.DataContext).OnLoad();
    }

    private void OnApplicationExit(object sender, ExitEventArgs e)
    {
        ioc.Release();
    }

    private void CreateMappings()
    {
        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();
    }
}
```

示例中的程序入口代码现在变得更简单了，所有控制反转容器注册动作都封装在一个专门的类中。代码清单9-10展示了任务列表应用程序中使用的IocConfiguration类。当程序退出时，

你可以在相应事件的处理器方法中调用该类的Release方法。

代码清单9-10　下面的配置类具有针对注册、解析和释放模式的全部三个阶段的方法

```
public class IocConfiguration
{

    private readonly IUnityContainer container;
    public IocConfiguration()
    {
        container = new UnityContainer();
    }

    public void Register()
    {
        container.RegisterType<ISettings, ApplicationSettings>();
        container.RegisterType<IObjectMapper, MapperAutoMapper>();
        container.RegisterType<ITaskService, TaskServiceAdo>();
        container.RegisterType<TaskListController>();
        container.RegisterType<TaskListView>();
    }

    public Window Resolve()
    {
        return container.Resolve<TaskListView>();
    }

    public void Release()
    {
        container.Dispose();
    }
}
```

示例中的Register方法和重构前应用程序入口的代码一样。但是，随着注册需求的增长，与在入口代码中直接和容器实例交互相比，将代码重构为多个方法能够让代码结构和意图更加清晰。

Resolve方法会返回一个通用的Window类对象，而Window类对象是WPF应用程序的公共解析根。在这里，返回的是TaskListView类实例，因为它是这个程序的主窗口。在诸如ASP.NET等其他应用程序类型中，通常会有多个解析根，每个控制器都有一个。MVC和其他模式应用程序的组合根会在本章后面详细讨论。

9

2. 命令式与声明式注册

到此为止，所有的注册代码都是命令式地从一个容器对象上调用方法。命令式注册方式的优势有：易读，比较简洁，编译时检查问题的代价非常小（比如防止代码输入错误等）。它的劣势是：注册过程的实现在编译时就已经固定了。如果你想替换现有实现，就必须要直接修改源代码并重新编译。

如果通过XML配置进行声明式注册，你就不再需要重新编译，只需要应用程序重新加载更新的配置即可。代码清单9-11展示了Unity的XML注册方式。

代码清单9-11　应用程序配置文件中的某个节可以描述接口应该如何映射到实现

```
<configuration>
  <configSections>
    <section name="unity"
             type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
  Microsoft.Practices.Unity.Configuration" />
  </configSections>
  <appSettings>
    <add key="TaskDatabaseConnectionString" value="Data Source=(local);Initial
  Catalog=TaskDatabase;Integrated Security=True;Application Name=Task List Editor" />
  </appSettings>
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <typeAliases>
      <typeAlias alias="ISettings" type="ServiceInterfaces.ISettings, ServiceInterfaces"
  />
      <typeAlias alias="ApplicationSettings" type="UI.ApplicationSettings, UI" />
      <typeAlias alias="IObjectMapper" type="ServiceInterfaces.IObjectMapper,
  ServiceInterfaces" />
      <typeAlias alias="MapperAutoMapper" type="UI.MapperAutoMapper, UI" />
      <typeAlias alias="ITaskService" type="ServiceInterfaces.ITaskService,
  ServiceInterfaces" />
      <typeAlias alias="TaskServiceAdo" type="ServiceImplementations.TaskServiceAdo,
  ServiceImplementations" />
      <typeAlias alias="TaskListController" type="Controllers.TaskListController,
  Controllers" />
      <typeAlias alias="TaskListView" type="UI.TaskListView, UI" />
    </typeAliases>
    <container>
      <register type="ISettings" mapTo="ApplicationSettings" />
      <register type="IObjectMapper" mapTo="MapperAutoMapper" />
      <register type="ITaskService" mapTo="TaskServiceAdo" />
    </container>
  </unity>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
</configuration>
```

示例XML展示了WPF任务列表应用程序的配置文件内容。为Unity增加的配置节包括`typeAlias`和`container`元素。前者用于为长的类型名称指定简短的别名，完整的类型名称需要包括程序集限定信息，这样Unity在运行时才可以找到指定的类型名。指定类型的别名后，`container`元素内会执行与`Register`方法一样的动作：建立接口和相应实现之间的映射关系。

为了使用XML配置文件，应用程序的入口代码也需要做些改动。代码清单9-12中高亮展示了这些不大的更改。

代码清单9-12　现在注册阶段要做的只是把配置文件中的相关节传递给容器对象

```
public partial class App : Application
{
    private IUnityContainer container;
```

```
    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        var section = (UnityConfigurationSection)ConfigurationManager.GetSection("unity");
        container = new UnityContainer().LoadConfiguration(section);

        MainWindow = container.Resolve<TaskListView>();
        MainWindow.Show();

        ((TaskListController)MainWindow.DataContext).OnLoad();
    }

    private void CreateMappings()
    {
        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();
    }
}
```

示例中的改动只有两行。首先你要使用ConfigurationManager类从配置文件加载unity节数据。节数据会被转换为UnityConfigurationSection类的实例，这样就可以把它传递给新的UnityContainer类实例的LoadConfiguration方法。完成这些后，就可以像前面一样，使用容器来解析应用程序的主窗口了。

尽管声明式的注册能将接口和相应实现的映射动作推迟到配置时，但它也有一些明显的缺陷，在很多情况下也不适合使用。它的最大缺陷就是太繁琐。当前示例已经很小了，但是依然有那么多类型需要定义别名和映射。某些情况下，需要注册的类型数目会是示例代码的好几倍，甚至更多，相应的XML配置文件也会变得非常大。XML文件中的别名定义和关系映射节中的输入错误直到运行时才能被发现和捕获，而命令式地注册代码能在编译时就检查到对应的问题。

声明式的注册不够好的另外一个显著原因是大多数控制反转容器都提供了很丰富的注册方式。比如lambda工厂，它会在解析接口时调用注册时提供的lambda方法。而这些高级的注册方式在声明式的XML配置文件中是无法做到的。

9

3. 对象的生命周期

要知道应用程序中不是所有对象的生命周期都是一样的，这一点很重要。也就是说，某些对象可能会比其他对象有更长的生命周期。当然，在.NET托管语言的上下文中，没有能直接销毁对象的方法，但是如果对象实现了IDispose接口，你就可以通过调用该接口的Dispose方法来释放该对象占有的相关资源。

比如，代码清单9-3展示的TaskService类中手动创建了一个SqlConnection实例，这是一个需要处理的代码味道。因为SqlConnection实例和TaskService实例的生命周期并不一致，后者的生命周期从应用程序启动时开始，一直到应用程序退出。如代码清单9-13所示，如果把SqlConnection注入到TaskService后，它会在应用程序运行时一直存在。然而，这并不意味着连接在此期间就一直打开着，因为打开连接是另外一个独立的操作。

代码清单9-13　有些资源的生命周期需要谨慎管理

```
private void OnApplicationStartup(object sender, StartupEventArgs e)
{
    CreateMappings();

    container = new UnityContainer();
    container.RegisterType<ISettings, ApplicationSettings>();
    container.RegisterType<IObjectMapper, MapperAutoMapper>();
    container.RegisterType<ITaskService, TaskServiceAdo>(new InjectionFactory(c => new
  TaskServiceAdo(new
  SqlConnection(c.Resolve<ISettings>().GetSetting("TaskDatabaseConnectionString")))));
    container.RegisterType<TaskListController>();
    container.RegisterType<TaskListView>();

    MainWindow = container.Resolve<TaskListView>();
    MainWindow.Show();

    ((TaskListController)MainWindow.DataContext).OnLoad();
}
// . . .
public class TaskServiceAdo : ITaskService
{

    private readonly IDbConnection connection;
    public TaskServiceAdo(IDbConnection connection)
    {
        this.connection = connection;
    }

    public IEnumerable<TaskDto> GetAllTasks()
    {
        var allTasks = new List<TaskDto>();

        using (connection)
        {
            connection.Open();

            using (var transaction = connection.BeginTransaction())
            {
                var command = connection.CreateCommand();
                command.Transaction = transaction;
                command.CommandType = CommandType.StoredProcedure;
                command.CommandText = "[dbo].[get_all_tasks]";

                using (var reader =
  command.ExecuteReader(CommandBehavior.CloseConnection))
                {
                    while (reader.Read())
                    {
                        allTasks.Add(
                            new TaskDto
                            {
                                ID = reader.GetInt32(IDIndex),
```

```
                                Description = reader.GetString(DescriptionIndex),
                                Priority = reader.GetString(PriorityIndex),
                                DueDate = reader.GetDateTime(DueDateIndex),
                                Completed = reader.GetBoolean(CompletedIndex)
                            }
                        );
                    }
                }
            }
        }

        return allTasks;
    }

}
```

示例中应用程序入口代码处的第一个改动是使用了一个注入工厂来创建任务服务。这个lambda表达式通过容器解析参数后返回了一个新的服务实例。原来对ISetting接口的GetSettings方法的调用也移到了lambda表达式内，用来获得连接字符串。这个字符串会传递给SqlConnection类的构造函数，而创建好的SqlConnection实例则会传递给TaskServiceAdo类的构造函数。

GetAllTasks方法中的using(connection)语句是有问题的。using语句结束前会确保调用SqlConnection类的Dispose方法。这样，在调用GetAllTasks方法后，连接就已经变得无效了，因为它占有的资源已经被释放了。想要使用连接，你能做的只能是再次调用GetAllTasks方法。

假设TaskServcieAdo类也实现了IDisposable接口，那么在它的Dispose方法中再去调用连接实例的Dispose方法会如何？代码清单9-14展示了这种方式。

代码清单9-14　服务实现了IDisposable接口，因为它可以去释放连接的资源

```
public class TaskServiceAdo : ITaskService, IDisposable
{
    public TaskServiceAdo(IDbConnection connection)
    {
        this.connection = connection;
    }

    public IEnumerable<TaskDto> GetAllTasks()
    {
        var allTasks = new List<TaskDto>();

        connection.Open();

        try
        {
            using (var transaction = connection.BeginTransaction())
            {
                var command = connection.CreateCommand();
                command.Transaction = transaction;
                command.CommandType = CommandType.StoredProcedure;
```

```
                command.CommandText = "[dbo].[get_all_tasks]";

                using (var reader =
command.ExecuteReader(CommandBehavior.CloseConnection))
                {
                    while (reader.Read())
                    {
                        allTasks.Add(
                            new TaskDto
                            {
                                ID = reader.GetInt32(IDIndex),
                                Description = reader.GetString(DescriptionIndex),
                                Priority = reader.GetString(PriorityIndex),
                                DueDate = reader.GetDateTime(DueDateIndex),
                                Completed = reader.GetBoolean(CompletedIndex)
                            }
                        );
                    }
                }
            }
        }
        finally
        {
            connection.Close();
        }

        return allTasks;
    }

    public void Dispose()
    {
        connection.Dispose();
    }
}
```

上面的示例并不是在GetAllTasks方法中释放连接，而是在释放服务本身时才释放它。这就牵扯到何时释放任务服务这个重要的问题。需要在构造函数注入ITaskService接口实例的所有客户端类型也都要实现IDisposable接口吗？谁来释放这些对象？最终，你总是需要在某些地方调用Dispose方法。

如果一个类通过构造函数得到了一个依赖项，它就不应该手动释放该依赖项的资源。这一点非常重要。因为该类无法确保该依赖项实例是否也同时提供给了其他类，因此手动释放它的资源会很可能会影响其他使用这个依赖的类。

使用依赖注入时管理对象生命周期问题的方式和原始服务实现的方式很接近。

- 连接工厂

工厂模式就是一种通过委托一个专门用来创建对象的类来替代手动实例化对象过程的方式。

代码清单9-15展示了可能的连接工厂接口定义。这个接口中的CreateConnection方法返回的是更加通用的IDbConnection接口实例，而不是要求所有客户端都要使用的SqlConnection类。

代码清单9-15　连接工厂接口看起来很简单

```
public interface IConnectionFactory
{
    IDbConnection CreateConnection();
}
```

可以把这个接口的实例注入到任务服务中，然后通过它来获取连接，而不再需要手动创建连接，这样也保持了模拟该任务服务实现的可测性。代码清单9-16展示了重构后的服务。

代码清单9-16　依赖注入可以与工厂模式协同工作

```
public class TaskServiceAdo : ITaskService
{
    private readonly IConnectionFactory connectionFactory;

    public TaskServiceAdo(IConnectionFactory connectionFactory)
    {
        this.connectionFactory = connectionFactory;
    }

    public IEnumerable<TaskDto> GetAllTasks()
    {
        var allTasks = new List<TaskDto>();

        using(var connection = connectionFactory.CreateConnection())
        {
            connection.Open();

            using (var transaction = connection.BeginTransaction())
            {
                var command = connection.CreateCommand();
                command.Transaction = transaction;
                command.CommandType = CommandType.StoredProcedure;
                command.CommandText = "[dbo].[get_all_tasks]";

                using (var reader =
  command.ExecuteReader(CommandBehavior.CloseConnection))
                {
                    while (reader.Read())
                    {
                        allTasks.Add(
                            new TaskDto
                            {
                                ID = reader.GetInt32(IDIndex),
                                Description = reader.GetString(DescriptionIndex),
                                Priority = reader.GetString(PriorityIndex),
                                DueDate = reader.GetDateTime(DueDateIndex),
                                Completed = reader.GetBoolean(CompletedIndex)
                            }
                        );
                    }
                }
```

```
            }
        }

        return allTasks;
    }
}
```

注意，CreateConnection方法返回的连接实例会在using语句块结束时被释放，因为工厂生成的IDbConnection接口实例都实现了IDisposable接口。通过接口继承，可以让单个实现类同时满足多个接口的定义。

然而，是否每个实现一定需要实现每个接口？有时候是的，但是考虑到一个接口会有很多种实现，尤其是在原始版本后过了很长时间，这需要做一个大胆的假定。但是对于IDisposable接口而言，很难保证每个类都实现了它。

- 负责人模式

你可以只在真正需要释放的类上实现IDisposable接口，而不是劳心费力地要求所有实现都实现它。但是这也会产生一个问题。如果工厂方法返回的结果（也就是你的接口实例）并没有实现IDisposable接口，你就无法使用using语句块来释放它。这种情况下，你必须使用负责人模式。

代码清单9-17中使用try/finally语句块代替了using语句块，同时，你可以在运行时检查实例对象是否实现了IDisposable接口。

代码清单9-17　负责人模式可以确保正确地释放对象占有的资源

```
public IEnumerable<TaskDto> GetAllTasks()
{
    var allTasks = new List<TaskDto>();

    var connection = connectionFactory.CreateConnection();
    try
    {
        connection.Open();

        using (var transaction = connection.BeginTransaction())
        {
            var command = connection.CreateCommand();
            command.Transaction = transaction;
            command.CommandType = CommandType.StoredProcedure;
            command.CommandText = "[dbo].[get_all_tasks]";

            using (var reader = command.ExecuteReader(CommandBehavior.CloseConnection))
            {
                while (reader.Read())
                {
                    allTasks.Add(
                        new TaskDto
                        {
                            ID = reader.GetInt32(IDIndex),
```

```
                                Description = reader.GetString(DescriptionIndex),
                                Priority = reader.GetString(PriorityIndex),
                                DueDate = reader.GetDateTime(DueDateIndex),
                                Completed = reader.GetBoolean(CompletedIndex)
                            }
                        );
                    }
                }
            }
        }
        finally
        {
            if(connection is IDisposable)
            {
                var disposableConnection = connection as IDisposable;
                disposableConnection.Dispose();
            }
        }

        return allTasks;
    }
```

示例中，只有实现了IDisposable接口的连接实例才会调用Dispose方法。无论工厂方法返回的结果是否实现了IDisposable，GetAllTasks方法都可以正常工作。对于实现了IDisposable接口的实例，它所占有的资源就会被正确释放。

负责人模式会明确地释放实现了IDisposable接口的实例对象，从而有效地忽略那些没有实现IDisposable接口的对象。但是，SOLID代码通常会有很多修饰器存在，它们会逐层包装以提供额外的功能。这种情况下，如果顶层对象实现了IDisposable接口，负责人模式是可以正常工作的。但是如果外部修饰器对象没有实现IDisposable接口，而内层对象实现了IDisposable时，负责人模式就没有办法正确释放这些内层的实例了。此时，你必须应用工厂隔离模式。

● 工厂隔离模式

这种模式能够明确地释放整个复杂的对象图，而SOLID代码通常会形成这样的对象图。这个模式的名称来源于图书馆常用的带手套的箱子。这些玻璃或金属的箱子会自带手套以确保人们对箱内内容的操作是安全的。类似地，工厂隔离模式能够保证安全访问对象的实例，而且这些实例会在使用后被正确地释放。

只有在接口没有实现IDisposable时才需要应用工厂隔离模式。要求所有类都实现IDisposable接口的Dispose方法是不现实的，也是不必要的。相反，IDisposable应该被看作实现细节并由各个类自己做主是否实现它。这就是负责人模式和工厂隔离模式的应用场景。

前面的示例都在围绕着IDbConnection接口实例的生命周期进行讲解，但该接口实际上已经继承了IDisposable接口。那么，如果我们假设这个接口并没有扩展继承IDisposable接口，从客户端代码角度看到的工厂隔离模式代码就会如代码清单9-18所示。

代码清单9-18　一个使用工厂隔离模式的客户端代码示例

```
public IEnumerable<TaskDto> GetAllTasks()
{
    var allTasks = new List<TaskDto>();
    connectionFactory.With(connection =>
    {
        connection.Open();
        using (var transaction = connection.BeginTransaction())
        {
            var command = connection.CreateCommand();
            command.Transaction = transaction;
            command.CommandType = CommandType.StoredProcedure;
            command.CommandText = "[dbo].[get_all_tasks]";
            using (var reader = command.ExecuteReader(CommandBehavior.CloseConnection))
            {
                while (reader.Read())
                {

                    allTasks.Add(
                        new TaskDto
                        {
                            ID = reader.GetInt32(IDIndex),
                            Description = reader.GetString(DescriptionIndex),
                            Priority = reader.GetString(PriorityIndex),
                            DueDate = reader.GetDateTime(DueDateIndex),
                            Completed = reader.GetBoolean(CompletedIndex)
                        }
                    );
                }
            }
        }
    });

    return allTasks;
}
```

工厂隔离模式并没有使用常见的返回工厂产品的实例的`Create`方法，而是使用`With`方法，该方法可以接受一个以工厂产品为参数的lambda方法。

这样做带来的好处是：工厂方法返回实例的生命周期会显式地由lambda方法决定。这会让无法控制对象生命周期的客户端代码变得非常简练。工厂实现本身非常简单，如代码清单9-19所示。

代码清单9-19　创建一个隔离工厂很简单

```
public class IsolationConnectionFactory : IConnectionIsolationFactory
{
    public void With(Action<IDbConnection> do)
    {
        using(var connection = CreateConnection())
        {
            do(connection);
        }
```

```
    }
}
```

其中的`With`方法能够创建带有大量修饰器、适配器和组合（也是SOLID建议的）的复杂对象图，而且可以管理这些对象的生命周期，客户端代码无需操心任何资源释放事宜，只需要简单地使用这些对象即可。

注意，如果把lambda方法范围内的实例对象赋值给一个有更长生命周期的变量，那么工厂隔离模式就会失效，所以并不鼓励在客户端代码中做这样的赋值。

9.2 比较复杂的注入

通过使用各种不同的框架，依赖注入的实现方式也可以有很多种。有些模式有着积极的作用，它们能够在确保完成目标的同时支持和增强依赖注入的能力。其他一些模式则相反，它们会背离依赖注入的根本目的并且会削弱依赖注入的能力。

有两个这样的模式特别值得注意。第一个是服务定位器反模式，不幸的是，它也很常见。它在很多框架和库中都有应用，有时候，它也是唯一能提供依赖注入钩子的方式。比服务定位器更糟糕的一个反模式是*非法注入*（Illegitimate Injection），它的名称并没有充分表明它的副作用。它有时会使用依赖注入的灰色地带，能够在不恰当地提供依赖的情况下构建服务、控制器和其他一些类似的实体对象。

当你在使用依赖注入时，不同类型的应用程序会需要不同的设置。不论是哪种类型的应用程序，你都需要识别组合根以便在正确的地方集成你的注册代码。WPF应用程序和Windows Form应用程序的组合根是不同的，而二者也都与ASP.NET MVC应用程序的组合根不同。

在一些高级场景中，无论是通过穷人的依赖注入手动组合类型，还是使用一个控制反转容器的单个注册类型，这两种方式都太繁琐且费时费力。通过一个或多个约定推迟注册的过程，你能够消除很多没用的样板代码，但同时也能提供一些手动的注册来处理那些不满足约定的边界情况。

9.2.1 服务定位器反模式

服务定位器看起来与控制反转容器很相似，这也正是它们经常不会被怀疑给代码造成破坏的原因。代码清单9-20展示了Microsoft的模式和实践团队提供的一个服务定位器的示例。

代码清单9-20 `IServiceLocater`接口看起来就像是另外一种形式的控制反转容器

```
public interface IServiceLocator : IServiceProvider
{
    object GetInstance(Type serviceType);

    object GetInstance(Type serviceType, string key);

    IEnumerable<object> GetAllInstances(Type serviceType);
```

```
        TService GetInstance<TService>();

        TService GetInstance<TService>(string key);

        IEnumerable<TService> GetAllInstances<TService>();
    }
```

其中诸如TService GetInstance<TService>()之类的方法能够与IUnityContainer接口中的定义直接对应起来，只是后者使用的方法名为Resolve。问题出在服务定位器的使用方式上，代码清单9-21展示的静态ServiceLocator类暴露了这个问题。

代码清单9-21　这个静态类就是将服务定位器归类为反模式的根本原因

```
    /// <summary>
    /// This class provides the ambient container for this application. If your
    /// framework defines such an ambient container, use ServiceLocator.Current
    /// to get it.
    /// </summary>
    public static class ServiceLocator
    {
        private static ServiceLocatorProvider currentProvider;

        public static IServiceLocator Current
        {
            get { return currentProvider(); }
        }

        public static void SetLocatorProvider(ServiceLocatorProvider newProvider)
        {
            currentProvider = newProvider;
        }
    }
```

我在示例中保留的摘要注释直接点出了问题所在。环境容器（ambient container）的概念已经透露了有一个容器存在的细节信息。尽管把具体的服务定位器实现隐藏在接口之后是正确的，但是问题在于它在任意类型内而不只是在组合根内暴露了服务定位器或者容器存在的信息。代码清单9-22显示了重写TaskListController以使用ServiceLocator时TaskListController的样子。

代码清单9-22　服务定位器允许类检索任何内容，无论是否适合

```
public class TaskListController : INotifyPropertyChanged
{
    public void OnLoad()
    {
        var taskService = ServiceLocator.Current.GetInstance<ITaskService>();
        var taskDtos = taskService.GetAllTasks();
        var mapper = ServiceLocator.Current.GetInstance<IObjectMapper>();
```

```
            AllTasks = new
        ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
        }

        public ObservableCollection<TaskViewModel> AllTasks
        {
            get
            {
                return allTasks;
            }
            set
            {
                allTasks = value;
                PropertyChanged(this, new PropertyChangedEventArgs("AllTasks"));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged = delegate { };

        private ObservableCollection<TaskViewModel> allTasks;
    }
```

这个示例没有构造函数，当然也就没有构造函数注入。相反，该类在需要的地方直接调用静态的ServiceLocator类并返回请求的服务。记住，像这样的静态类都是天钩（skyhook），它是一种代码味道。

更糟糕的是，该类能从服务定位器检索任意对象。这样你就违背了依赖注入的“好莱坞准则”：不要调用我们，我们会调用你。相反，你是在直接要求需要的东西，而不是从其他地方传递得来的。你又如何知道这个类到底还需要什么样的依赖呢？使用服务定位器，你必须检查代码以搜索变化无常的调用，这些调用会检索某个需要的服务。你只需要看一眼构造函数或者智能感知列出的信息，就可以从构造函数注入中清楚地看到所有的依赖。

服务定位器模式并没有单元测试的问题。因为在使用它之前，你可以设置一个IServiceLocator接口的实现，也就是说，可以模拟服务定位器来对使用它的其他类型做单元测试。至少，服务定位器模式并没有阻止你去做单元测试。只是大量的注册接口和相应实现类映射关系的代码有些不合理，因为控制器、服务器和其他类的实现代码会被这些基础代码污染。在没有需要解决的问题时应用这种服务定位器模式会更加不合理，而构造函数注入并不需要担心这些问题。

Unity中提供了一个服务定位器的适配器。代码清单9-23展示了如何通过它注册设置好的映射关系。

代码清单9-23　服务定位器会直接委托UnityContainer实例来解析实例对象

```
    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        container = new UnityContainer();
        container.RegisterType<ISettings, ApplicationSettings>();
```

```
        container.RegisterType<IObjectMapper, MapperAutoMapper>();
        container.RegisterType<ITaskService, TaskServiceAdo>();
        container.RegisterType<TaskListController>();
        container.RegisterType<TaskListView>();

        ServiceLocator.SetLocatorProvider(() => new UnityServiceLocator(container));

        MainWindow = container.Resolve<TaskListView>();
        MainWindow.Show();

        ((TaskListController)MainWindow.DataContext).OnLoad();
    }
```

这与前面的示例看起来很像，只是没有去设置定位器提供者。但是，对Resolve方法的调用并没有真正解析对象图；也不再有任何依赖注入到TaskListView类的构造函数中，所有的依赖都是在需要时才在该类的方法中单独获取的。

服务定位器模式是个很好的反面例子，它表面声称的东西并不符合应用后的实际效果。它声称带有默认构造函数的类没有依赖，但是显然不是这样的：它们肯定有依赖，要不然你为何要从服务定位器中获取它们。

不幸的是，有时又必须应用服务定位器反模式。在某些应用程序类型里，特别是Windows Workflow Foundation，基础库根本没有从构造函数注入的任何机会。在这些情况下，你的唯一选择就是服务定位器，它至少比完全不注入依赖要好。虽然我对反模式提出了这么多批判，但是它们肯定比完全手动构造依赖要好。毕竟，它也能够使用接口提供所有重要的扩展点，也就是说，可以获得修饰器、适配器以及其他一些类似的好处。

注入容器

与服务定位器密切相关的是在类型中注入容器的概念。同样，这把类变成了安全的关键点，通过这种方式将容器注入到类之后，就可以自由使用容器获取任何想获取的实例对象。假设有这样一个类，它的多个方法中零零散散地获取了很多服务对象实例。再假设另外一个类是从构造函数中注入了同样多的依赖对象，该类会在构造函数入口处对这些依赖对象做完整的前置条件检查并会在发现空引用时引发异常。显然，这两个类都做得太多了，如果需要那么多的依赖，就应该把它们重构为规模更小的类或者将各种依赖组织为有意义的修饰器。但是，相比较假设的第一个类，只有第二个类才能明显地暴露出这种代码味道，这样才能有机会尽早发现并消除它。

另外，从构造函数注入容器的类也必须引用容器的程序集。这会让容器基础代码扩散到整个代码库中，因为每个类都接受了注入的容器以获取它们真正需要的服务对象。

9.2.2　非法注入

非法注入表面看起来很像正常的，就像正确实现的依赖注入。它们也有可以注入依赖的构造函数，也是通过穷人的依赖注入或控制反转容器提供依赖对象。

但是，由于带有默认构造函数，这些对穷人的依赖注入和控制反转容器的应用已经的的确确

被破坏了。代码清单9-24展示了这种情况，默认构造函数中直接构造了一些依赖的实现对象，因此也绕开了依赖注入。

代码清单9-24　构造函数直接引用实现会直接让依赖注入的很多优势失效

```
public class TaskListController : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged = delegate { };

    private readonly ITaskService taskService;
    private readonly IObjectMapper mapper;
    private ObservableCollection<TaskViewModel> allTasks;

    public TaskListController(ITaskService taskService, IObjectMapper mapper)
    {
        this.taskService = taskService;
        this.mapper = mapper;
    }

    public TaskListController()
    {
        this.taskService = new TaskServiceAdo(new ApplicationSettings());
        this.mapper = new MapperAu toMapper();
    }

    public void OnLoad()
    {
        var taskDtos = taskService.GetAllTasks();
        AllTasks = new
  ObservableCollection<TaskViewModel>(mapper.Map<IEnumerable<TaskViewModel>>(taskDtos));
    }

    public ObservableCollection<TaskViewModel> AllTasks
    {
        get
        {
            return allTasks;
        }
        set
        {
            allTasks = value;
            PropertyChanged(this, new PropertyChangedEventArgs("AllTasks"));
        }
    }

}
```

这就意味着，该类必须引用实现所在的程序集，并且同时引入它的整个依赖链。这不就是随从反模式吗？尽管第一个构造函数是接受注入的接口实例，看起来很好地应用了阶梯模式和正确的依赖注入，但是第二个默认构造函数却直接破坏了它们带来的好处。

如果默认实现不再是你想要的时怎么办？该类会被修改为更喜欢的类。如果一个默认构造函

数不够用，那么你想在某些场景下实现A，而在其他场景下又想实现B时，该怎么办？构造函数带来的副作用会很快让人失去耐心的。

有时这种容器注入的反模式也会被用来支持单元测试。要模拟测试类的默认实现看起来并没有依赖，它们可能就在该类的内部。类内部不应该包含任何只用于支持单元测试的代码。很常见的例子就是，先把`private`方法变为`internal`，然后应用`InternalsVisibleToAttribute`属性来让测试程序集访问这些方法，而不是只让测试类通过`public`接口进行测试。实话实说，依赖注入支持单元测试的能力也存在被夸大的现象，但是这恰好说明了关键点所在：你已经通过使用接口来支持单元测试并将它们注入到了构造函数，因此模拟对象可以（也应当）通过构造函数注入到接口的类实现中。

非法注入被归类为反模式并不会因为默认构造函数的可见性而改变。无论默认构造函数是`public`、`protected`、`private`或者`internal`，事实都很清楚：你在引用它时不应该直接引用具体的实现。

9.2.3　组合根

应用程序中只应该有一个位置知道依赖注入的细节，这个位置就是组合根。在使用穷人的依赖注入时就是你手动构造类的地方，在使用控制反转容器时就是你注册接口和实现类间映射关系的地方。

理想情况下，组合根和应用程序的入口越近越好。这样能让你尽快配置好依赖注入。组合根提供了一个查找依赖注入配置的公认位置，它能帮你避免把对容器的依赖扩散到应用程序的其他地方。这也意味着不同种类的应用程序有着不同的组合根。

1. 解析根

与组合根密切相关的一个概念是解析根。它是要解析的目标对象图中根节点的对象类型。在前面的WPF示例中，解析根甚至可以是个单例对象，但是通常情况下，它们都是一组基于公共基类的类型。

有些情况下，你要自己手动获得解析根，但是在有些类型的应用程序已经利用了依赖注入（比如MVC模式）来处理解析根时，你需要做的只是注册接口和类的映射关系。

2. ASP.NET MVC

MVC模式的工程已经很好地通过控制反转容器应用了依赖注入。这些工程已经清楚地定义了解析根和组合根，也能够轻易地扩展以支持你可能需要为控制反转容器集成的任何库。

MVC应用程序的解析根就是控制器。所有来自浏览器的请求都会被路由到被称为动作（action）的控制器方法上。每当请求到来时，MVC框架会将URL映射为某个控制器名称，然后找到名称对应的类并实例化它，最后再在该实例上触发动作。图9-4中的UML时序图展示了这个交互的过程。

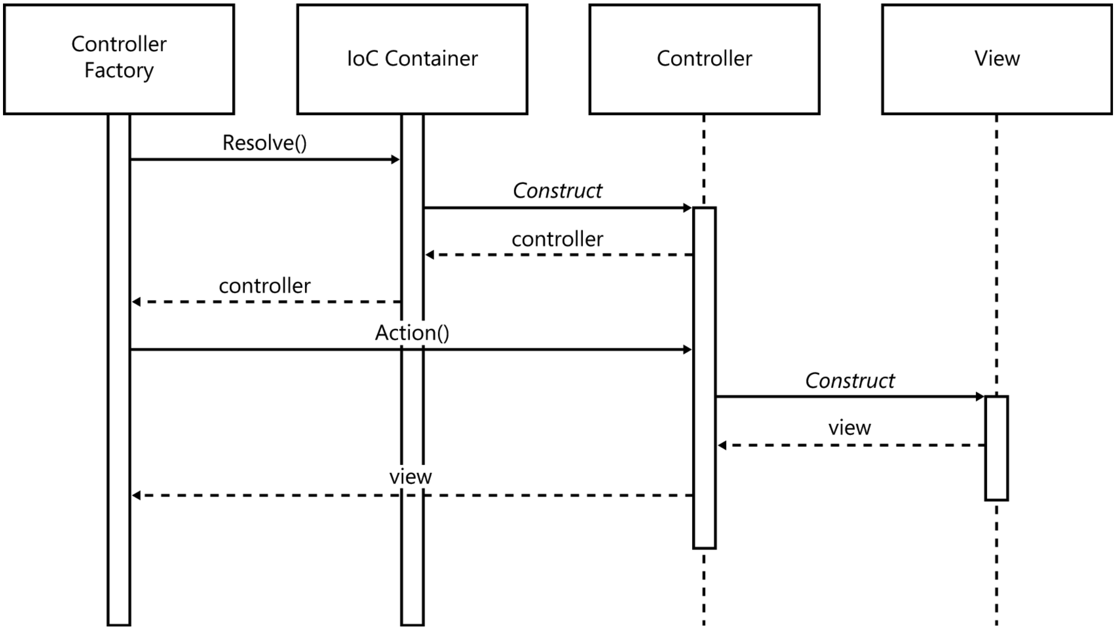


图9-4 UML时序图展示了MVC通过工厂构造控制器的过程

在使用控制反转容器进行依赖注入时，更确切地讲，实例化控制器的过程就是解析（resolution）控制器的过程。这就意味着，你能轻易地按照注册、解析和释放的模式，最小化对`Resolve`方法的调用，理想情况下，就只应该在一个地方调用该方法。

代码清单9-25展示了任务列表应用的ASP.NET MVC用户界面的组合根。

代码清单9-25 `HttpApplication`的`Application_Start`方法是Web应用中一个常见的组合根

```
public class MvcApplication : HttpApplication
{
    public static UnityContainer Container;

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);

        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();

        Container = new UnityContainer();
        Container.RegisterType<ISettings, ApplicationSettings>();
        Container.RegisterType<IObjectMapper, MapperAutoMapper>();
        Container.RegisterType<ITaskService, TaskServiceAdo>();
        Container.RegisterType<TaskViewModel>();
```

```
            Container.RegisterType<TaskController>();

            ControllerBuilder.Current.SetControllerFactory(new
      UnityControllerFactory(Container));
        }
    }
```

示例中的一些代码就是创建新MVC应用时的默认模板代码，它包括了所有MVC模式相关的初始化代码，比如注册域、过滤、路由和绑定等。这些代码都要尽早在Application_Start方法中执行。ASP.NET应用在IIS中启动后，第一个收到的请求会触发调用Application_Start方法。后置代码文件Global.asax包含了HttpApplication类的特定于应用的子类，Application_Start方法也位于该子类的实现中。

上面的示例中，除了针对MVC的TaskController类外，其余的服务接口和实现都是可以直接重用的。前面一节中的TaskController类是针对WPF编写的，因此也不可以在WPF以外的上下文中重用，需要编写新的控制器类。但是，新的TaskController类也和WPF版本的控制器类做了很多相同的工作，包括获取任务数据，使用IObjectMapper将任务数据转换为视图友好的格式，不同点只是基于MVC控制器的基类。因为TaskController类是继承System.Web.Mvc.Controller类的，所以它就是应用的解析根。代码清单9-26展示了这个新的TaskController类。

代码清单9-26　TaskController是一个解析根，它有一个需要依赖的构造函数

```
public class TaskController : Controller
{
    private readonly ITaskService taskService;
    private readonly IObjectMapper mapper;

    public TaskController(ITaskService taskService, IObjectMapper mapper)
    {
        this.taskService = taskService;
        this.mapper = mapper;
    }

    public ActionResult List()
    {
        var taskDtos = taskService.GetAllTasks();
        var taskViewModels = mapper.Map<IEnumerable<TaskViewModel>>(taskDtos);
        return View(taskViewModels);
    }
}
```

List方法是该类的动作方法，会被渲染所有任务数据的同一个视图类调用。在WPF应用中，控制器首先委托ITaskService来获取任务数据，然后使用IObjectMapper类将这些数据转换为视图模型定义的数据类型，以供视图使用。

> **注意**　此处使用的也是用于WPF的相同ViewModel。这样是可行的，因为INotify-PropertyChanged接口不仅仅用于WPF上下文（该接口位于System.ComponentModel命名空间）。然而，MVC并不在乎这个接口，也不会响应任何从ViewModel中引发的事件。此外，MVC允许你使用校验提示和其他MVC程序集中的类似属性来修饰ViewModel，因此，最好还是针对MVC上下文定义新的ViewModel。

默认情况下，MVC控制器需要一个公共的默认构造函数，这样MVC框架才可以在调用动作方法前构造控制器实例。但是，使用依赖注入时，你需要的是能接受所需服务接口参数的构造函数。幸运的是，MVC使用工厂模式创建控制器实例，这就为你的控制器实现提供了扩展点。如代码清单9-27所示。

代码清单9-27　MVC框架提供了很多扩展点，包括能够利用依赖注入的自定义控制器工厂

```
public class UnityControllerFactory : DefaultControllerFactory
{
    private readonly IUnityContainer container;

    public UnityControllerFactory(IUnityContainer container)
    {
        this.container = container;
    }

    protected override IController GetControllerInstance(RequestContext requestContext,
  Type controllerType)
    {
        if (controllerType != null)
        {
            var controller = container.Resolve(controllerType) as IController;
            if (controller == null)
            {
                controller = base.GetControllerInstance(requestContext, controllerType);
            }
            if (controller != null)
                return controller;
        }
        requestContext.HttpContext.Response.StatusCode = 404;
        return null;
    }
}
```

当你在Application_Start方法中构建UnityControllerFactory实例时，需要传入一个UnityContainer类参数。如示例中粗体高亮的代码所强调的，可以用GetControllerInstance方法代替容器的Resolve方法，前者可以根据类创建所需的控制器实例。这就是解析出控制器（解析根）的地方，同时能得到的还包括对象图中其他可能需要的对象。

需要记住的是，这个示例中实例对象的生命周期是不同的。应用启动时，控制反转容器就创建好了，映射关系也注册了。然而，直到有请求发生时才会解析控制器实例。每个请求到来时，

应用会解析得到相应的控制器实例，当请求处理结束后，该控制器实例就已经超出了自己的作用范围，因此不再需要它了。

3. Windows Forms

Windows Forms应用中引导依赖注入的方式更像WPF应用，而不是ASP.NET MVC应用。二者的解析根都是视图，视图的构造函数需要传入的是表示器或控制器参数，并由视图开始处理整个对象图。

代码清单9-28展示了Windows Forms任务列表应用前端的组合根。它位于`Program`类的`Main`方法中，该方法是Windows Froms应用的入口。按照惯例，让注册代码越接近应用的入口越好。

代码清单9-28 `Program`类的`Main`方法是应用的入口点，可以作为一个很好的组合根

```
static class Program
{
    public static UnityContainer Container;

    [STAThread]
    static void Main()
    {
        AutoMapper.Mapper.CreateMap<TaskDto, TaskViewModel>();

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        Container = new UnityContainer();
        Container.RegisterType<ISettings, ApplicationSettings>();
        Container.RegisterType<IObjectMapper, MapperAutoMapper>();
        Container.RegisterType<ITaskService, TaskServiceAdo>();
        Container.RegisterType<TaskListController>();
        Container.RegisterType<TaskListView>();

        var mainForm = Container.Resolve<TaskListView>();
        Application.Run(mainForm);
    }
}
```

注意，在这个Windows Forms应用中，你不仅可以重用服务的WPF版本实现，也可以重用`TaskListViewController`类，因为它并不依赖任何特定于WPF的东西。当然，将来很可能会有些特定于WPF的依赖，因此为支持Windows Forms平台创建专门的控制器或表示器是有必要的。

这个应用的视图非常简单，后置代码中只需要给构造函数传入控制器实例并实例化数据绑定，如代码清单9-29所示。当你在使用模型–视图–表示器模式时，视图会实现一个表示器，从而能够手动委托调用以设置数据的接口。

代码清单9-29 视图使用数据绑定将获得的任务列表设置到一个数据表格控件上

```
public partial class TaskListView : Form
{
    public TaskListView(TaskListController controller)
```

```
        {
            InitializeComponent();

            controller.OnLoad();
            this.taskListControllerBindingSource.DataSource = controller;
        }
    }
```

如果不应用已有平台来解析视图，你就必须先自己解析得到视图实例，然后再把它传递给Windows Forms应用的启动方法`Application.Run`。这是在应用只有一个主视图时唯一合适的解析点，通常桌面应用都属于这种情况。此外，可以使用视图实现的控制器或表示器创建对话框和其他子窗口。

9.2.4 约定优于配置

通过配置来注册接口和相应实现类之间的映射关系会很费力，而且随着时间的推移，也会变得繁琐冗长。相反，你可以使用约定来减少需要编写的代码量。

约定是一组指令，用于告诉容器如何自动完成接口到相应实现类的映射。容器接受输入的指令，而不是注册。理想情况下，容器处理输入指令所得到的输出与你手动完成的注册效果一样。

> **注意** 这里说的“优于”也可以解释为“可以替代”。因此，这一节的主题可以是“使用约定替代配置”。

代码清单9-30展示了前面的示例是如何使用约定进行注册的过程。

代码清单9-30 使用约定可以极大简化注册阶段的代码

```
private void OnApplicationStartup(object sender, StartupEventArgs e)
{
    CreateMappings();

    container = new UnityContainer();
    container.RegisterTypes(
        AllClasses.FromAssembliesInBasePath(),
        WithMappings.FromMatchingInterface,
        WithName.Default
    );

    MainWindow = container.Resolve<TaskListView>();
    MainWindow.Show();

    ((TaskListController)MainWindow.DataContext).OnLoad();
}
```

单个`RegisterTypes`方法完成了所有的注册过程。该方法用于给容器提供如何查找类以及将

它们映射到相应接口的指令。上面示例提供给容器的指令包括以下这些。

- 注册基本路径bin目录下所有程序集包含的类。
- 把这些类映射到符合类命名约定的接口上。这里的约定（convention）是指，名为Service的实现类的对应接口的名称应该是IService。
- 注册每个映射关系时使用默认值来命名映射。默认值为空代表了映射关系是未命名的。

按照这些指令，容器会枚举bin目录下的每个程序集中的每个公开的类，找到它实现的所有接口，并把它映射到其中那个符合自己命名规则（前缀为代表Interface的I）的接口，同时并不需要为映射关系命名。不难想象，这样注册的结果要比你手动注册生成的映射关系量要大的多。然而，更重要的是如何保证正确地注册类和接口的映射关系。这也是约定注册方式引入的新问题。

不可否认，约定注册的方式的确让代码简化了很多，但也只局限在代码量更少的层次上。通过配置注册，能够很容易地知道每个接口对应的实现，而且能确保注册是正确的。

RegisterTypes方法的第一个参数是要注册类的集合。静态AllClasses类提供的一些辅助方法能够通过一些常见的策略获得要注册类的集合。第二个参数是一个函数，它的输入参数是第一个参数获得的实现类集合，输出的是映射得到的对应接口集合。静态WithMapping类提供了一些辅助方法以多种策略来为每个类找到合适的接口。第三个方法是另外一个函数，它会为每个类上的映射关系返回一个名称。静态WithName类提供了两个命名选项：总是返回空（因此映射也就是未命名的）的Default和使用类名作为映射名称的TypeName。后者允许你根据类名称获取映射到的类实例，调用的语句为Resolve<IService>("MyServiceImplementation")。

当然，上面示例代码中的方法参数很通用，你可以使用任何符合参数签名要求的其他方法以反映你需要的约定。如代码清单9-31所示，约定注册方式的关键点就是用于查找类，建立类和接口之间的映射关系，以及为映射关系命名的约定。

代码清单9-31　约定可以按照你的需求进行定制

```
public partial class App : Application
{
    private void OnApplicationStartup(object sender, StartupEventArgs e)
    {
        CreateMappings();

        container = new UnityContainer();
        container.RegisterTypes(
            AllClasses.FromAssembliesInBasePath().Where(type =>
  type.Assembly.FullName.StartsWith(MatchingAssemblyPrefix)),
            UserDefinedInterfaces,
            WithName.Default
        );

        MainWindow = container.Resolve<TaskListView>();
        MainWindow.Show();

        ((TaskListController)MainWindow.DataContext).OnLoad();
    }
```

```
    private IEnumerable<Type> UserDefinedInterfaces(Type implementingType)
    {
        return WithMappings.FromAllInterfaces(implementingType)
            .Where(iface => iface.Assembly.FullName.StartsWith(MatchingAssemblyPrefix));
    }
}
```

这个示例不再是直接获取bin目录下所有程序集中的所有类，而是只查找那些符合指定前缀字符串的程序集。通常情况下会使用点分隔的命名方式，这样只需要去匹配名称中的顶层命名空间即可。所以，Microsoft.Practice.Unity是DLL名称，也是该DLL内所有类的命名空间。如果bin目录下有Microsoft.Practice.Unity这个DLL文件（如果你在使用Unity的话就一定会有），你也许想在检索并建立映射关系时直接忽略它。一种简单的办法就是只从符合应用自己的前缀的程序集中获取类。比如，你应该使用诸如MyBusiness或OurProject等来代替Microsoft作为文件名称前缀。

第二个参数已经被满足参数签名要求的本地方法`UserDefinedInterfaces`替代。给定一个实现类`Type`，该方法会将映射返回到该类的一个接口集合。这里也不需要自己编写特别复杂的代码，只需要调用`WithMappings.FromAllInterfaces`方法，该方法会返回指定类实现的所有接口。返回的接口集合中很可能包括你并不想要的一些接口，比如`INotifyPropertyChanged`或`IDataErrorInfo`等。所以，你还是应该只去检索那些符合你的命名前缀规则的程序集，这样可以确保只建立你自己的类和接口之间的映射关系。

1. 优缺点

和穷人的依赖注入以及控制反转容器进行注册类似，使用约定进行注册一样有优点也有缺点。你要写的代码量是更少了，但是同时代码也比其他方法中声明式的代码更难直接理解了。

约定在开始阶段的设置也更复杂。如果你在编写真正的SOLID代码，也不是所有类和接口之间的映射关系都是一对一的。实际上，只有一个实现（不包括用于单元测试的模拟实现）的接口的情况本身就是一个代码味道。通常情况下，一个接口都应该有多于一个的具体实现，不论它们是适配器、修饰器或是不同策略的实现等，而这种现状也会让按照约定注册变得更复杂和困难。注入类的对象图会变得更加复杂，所以很难整理出一个让类和接口相互映射的规则。在这种情况下，约定只会涵盖所需注册代码的一小部分，而不像一般情况那样涵盖所有注册代码。

Mark Seemann是*Dependency Injection in .NET*（2011年由Manning Publications出版）一书的作者，他已经探索出三种可用的依赖注入方式并得到了如图9-5所示的结果。简而言之，对三种方式的取舍有两个标准：价值和复杂度。价值用于衡量选项的作用和意义，从无意义到有价值的。复杂度用于衡量选项的难度，从简单的到复杂的。如图9-5所示，三种方式位于贝尔曲线的不同位置。穷人的依赖注入方式简单但很有价值，而约定优于配置的方式虽然复杂但也很有价值。因此，它们二者之间的主要区别在于，使用约定要比手动创建类和在这些类基础上构造出的类对象图要更复杂一些。

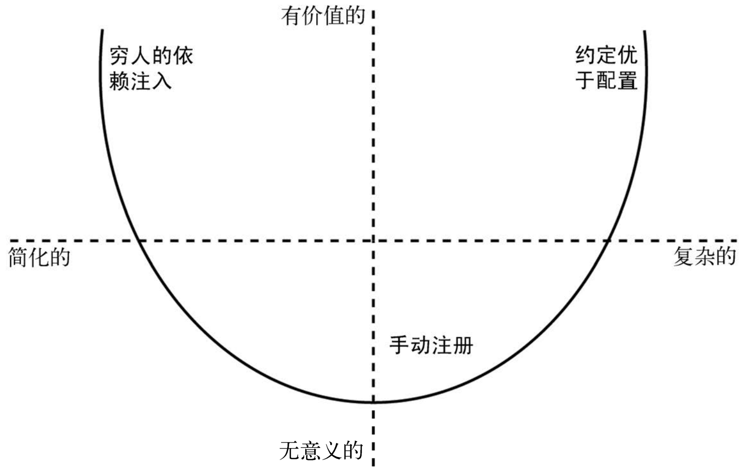


图9-5　象限图中三种依赖注入方式各有优缺点

很有意思的是，Seemann认为手动注册的复杂度适中，但并没有实用价值。为什么他这样认为呢？主要是因为使用容器手动注册类型是弱类型化的。如果你尝试把一个类传递给要求不同参数类型的实例时，编译器会在生成时给出报错。然而，如果你给控制反转容器传入不符合要求的类时，编译器并不会报错。相反，你只有在运行时才能看到出错信息，这会让你陷入一个编写、编译、运行和测试的死循环中。不仅如此，你还需要花费很多时间和精力学习如何使用容器注册映射关系，但实际上，这样做是得不偿失的。

现在，选择看起来简单了，要么选择穷人的依赖注入，要么选择约定。如果项目很简单并且只需要少量的映射关系，此时穷人的依赖注入很合适，只需要手动构造对象即可。如果项目变得更复杂，则会需要建立很多接口和类之间的映射关系，此时应该使用约定处理大多数的注册，其余的映射关系手动处理即可。

我也强烈推荐大家去阅读Mark Seemann的博客[①]，那里有很多他的方法论相关的主题，所有主题的讨论都是有条不紊的。

9.3　总结

依赖注入是将本书其他部分结合在一起的主线。没有依赖注入，就无法清楚地将依赖从类中分解出来，也无法将它们的实现隐藏在通用的接口后。这些扩展点对于自适应代码的编写而言非常关键，也是让逐步变大变复杂的应用稳步取得进展的关键。

实现依赖注入有多种不同的方式，每个都有它适合的场景。不论你是在使用穷人的依赖注入或者带有少量手动映射的约定，总是使用依赖注入要比具体的实现方式更重要。

实际上，有些常见的对依赖注入的滥用应该被归类到代码味道或反模式中。服务定位器和非

① blog.ploeh.dk。

法注入就是两种常见的滥用，它们会破坏正确应用依赖注入所得到的很多好处。

每个应用都有一个组合根和一个解析根，二者可以帮助你理解如何利用依赖注入来组织应用中的所有类。如果注册过程是应用初始化的一个主要问题，组合根就总是应该非常接近应用的入口位置。解析根则是唯一应该解析获取的对象类型。在某些应用中，解析根只有一个实例，而在其他一些应用中，解析根类会有一组不同的子类。

依赖注入对SOLID代码的编写影响很大，它看似简单，实则不然。在实际应用中，强大的依赖注入经常无法得到正确的理解和应用。

Part 3 第三部分

自适应实例

本部分内容

本书的这一部分会带你实践软件产品开发的几个初始阶段。这一部分包含的三章会从一个虚拟的团队和项目角度来讲解所有团队成员形成的约定以及他们如何按照约定作出必要的决定。

这一部分的代码示例会讲解如何选择在前两个部分中讲解过的模式和实践。虽然无法覆盖所有主题，但是也回答了很多常见的实现问题。

就像在本书的其他部分中一样，你可以从 GitHub 上下载非常好用的示例代码。有关如何使用 Git 进行源代码控制的简要介绍，请参见附录。

第 10 章 自适应实例简介

完成本章学习之后，你将学到以下技能。

- 了解将要开发自适应示例应用的团队。
- 理解该应用的产品特性。
- 在第零个冲刺为该应用创建好初始的产品积压工作。

这一部分的三个章节会按照Scurm流程以及前面章节讲解到的自适应设计原则逐步构建出一个可以工作的应用。这是本书至此所有内容的高潮部分，展示了如何构造出一个清晰的全景图。在阅读剩余章节的内容时，我推荐你也同时学习配套的代码[1]。没有本章的讲解，单独看代码会缺乏上下文信息。同样，如果没有完整的Microsoft Visual Studio解决方案源代码，只通过后面章节中摘取的示例代码清单也无法看到项目的全景。

本章的格式会尽量反映实际项目的场景，但有些地方会为了简洁和清晰而做出一些妥协。接下来会在本章介绍一个虚拟的Scrum团队以及要开发产品的概要。

10.1 Trey Research 公司

下面的示例应用将会由一个虚拟的名为Trey Research的公司开发，他们为自己具备编写出优秀自适应代码的能力而颇感自豪。

10.1.1 团队

示例应用会按照Scrum流程进行开发，所以需要成员担当Scrum团队中的各个角色。有关更详细的角色和Scrum流程介绍，请参见第1章。

团队包括了产品从创建到交付过程中需要的所有角色。产品负责人知道如何获取想要的应用特性，哪些特性优先级最高，哪些特性能为业务带来最大的收益。Scrum主管则负责团队使用的流程。他关注的是，流程要匹配团队，团队工作要没有障碍，用户故事开发过程中出现的问题要及时反馈给产品负责人。开发团队则包括了要实现故事的几个开发人员，和一个负责设计测试用例以及验证故事是否达到交付标准的测试分析人员。

① 有关如何访问示例应用的代码以及本书中的其余代码的说明，请参见附录。

1. 产品负责人

产品负责人Petra是一位非常资深的业务分析师，最近刚刚加入Trey Research公司。她非常擅长发现用户真正想要的东西，这就是产品负责人的一个杀手锏。她很大度地承认，她并不熟悉各种敏捷流程，但是非常乐意学习和实践产品负责人这个新角色。

在整个开发过程中，Petra一直都和客户公司保持联络以发现客户真正需要的东西和需要它们的理由。此外，她也会计算各种不同特性对于客户的价值，从而帮助开发团队更好地对工作项进行排序。

2. Scrum主管

Steve在公司同时兼任两个角色：Scrum主管和开发团队领导。公司打算很快修正这个兼任多个角色的问题，从而让Steve只从事他更喜欢的Scrum主管角色。为此，公司正计划聘任一个资深的开发人员作为专职的开发团队领导。

以Steve作为Scrum主管的能力，他可以确保团队遵守Scrum流程，让团队成员对流程的当前体现感到满意。他为自己能诚信透明地和任何指定的产品负责人或客户协作而自豪，因为他从不更改范围或随意承诺交付目标。

尽管Steve已经很难有时间去实际编写代码，但他仍然会参加设计会议并竭尽全力让整个开发团队朝着正确的方向前进。

3. 开发人员

David和Dianne是该公司的两个专职开发人员。David是一个初级开发人员，因为他是大学毕业后直接加入该公司的，而Dianne则处于中级水平。

Steve决定雇佣David的一个原因是他始终能坚持自学编程实践和技术。David总是如饥似渴地学习最前沿的开发技术。但是，他总是倾向于把每个新的技术看作灵丹妙药且会不加思考地应用在能用到的地方。这说明了David的实践能力很好，但是他的代码经常会包含大量根本不需要的中间层。

Dianne比David的经验更丰富，但是她已经厌倦了过去几年出现的技术浪潮。她也经历过David现在正在遇到的问题。Dianne也想竞聘开发团队领导的岗位，因此她也下决心要证明自己具备被提拔的资质。为此，她也非常想帮助David取得进步。

- 技术成熟度曲线

技术成熟度曲线是由一个名为Gartner的IT研究和咨询公司开发的。它是一个很好的用来评估新技能和技术的工具。如图10-1所示。

图中x轴表示时间的推进，y轴表示期望。最开始的是促动期，代表出现了一个新的技术发现或者一个有用的新技能或过程。不久后，期望值迅猛增长，直到达到“期望膨胀的顶峰”。在这个点上，那些新的技术或技能开始被认为并没有它们刚出现时那样强大。这也会导致后面期望值会允许跌落到“泡沫化的低谷期”。但是，这并不代表结束了，因为经过随后的“稳步爬升的光明期”会逐渐过渡到“实质产出的高峰期”。至此，新的技术或技能才真正确立了，并且大家对它的期望更加现实。

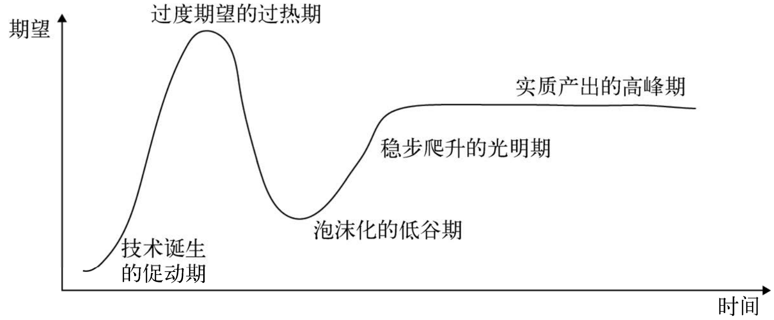


图10-1　技术成熟度曲线可以辅助解释对新技能和技术的态度

开发人员对一种或多种编程技术或技能的期望通常都会位于该技术成熟度曲线的某个位置。比如，在Trey Research公司里，David对涉及模式、SOLID原则、单元测试以及重构这些技术的期望处于过度期望的过热期。Dianne则刚刚进入泡沫化的低谷期。在本章后面的冲刺会议中，你要特别关注他们的问题和响应，这样才可以理解形成他们在技术成熟度曲线中所处相对位置上的原因。

为了完整起见，需要注意Steve的情况，他的丰富经验表明他已经经历了整个技术成熟度曲线，他目前处于实质产出的高峰期。不仅如此，Steve已经不会再像以前那样容易受到技术成熟度曲线的影响。他在面对那些听说可以增加生产力、质量和效率或其他度量值的新技能和技术时，会很谨慎。因此，在面对一个新的技能或技术时，他的期望值并不会出现大的波动。

4. 测试分析员

Tom是团队中的测试分析员。他主要负责测试自动化工作。他一直很擅长发现软件产品的缺陷以及通过逐步增加健壮性来改善应用的整体用户体验。他喜欢将特性看作黑盒，并不关心实现细节，只关心软件产品是否能够按照规格说明工作。

Tom自己觉得他在团队中的工作量过多，因为他要不停地返工很多已经做过的测试。也就是说，在一个故事达到验收标准前，他至少会测试两次。Tom很自豪自己的工作可以确保软件产品在客户机器上很少出现问题，因为他的自动化测试会早早发现它们。

10.1.2　产品

有个客户已经和Trey Research公司签订了合同，后者会为该客户开发一个新的名为Proseware的在线聊天应用程序。这是个基于网络的应用程序，它支持世界各地的用户同时在线聊天。在和客户初步沟通后，Petra知道客户对该项目有很多想法，但是客户决定一点一点地增加功能以便他们能够决定应用程序的开发方向。Trey Research公司非常适合该项目，因为该公司的开发风格就是增量的，也具备平衡频繁需求变更的能力。

在项目开始前，Petra同客户进行了交谈，这样她就可以给团队提供将要开发产品的相关信息。客户想要Trey Research托管Proseware这个产品，因此，团队可以自由选择他们认为对项目最合适的平台和工具。Petra也和客户确认了客户对Proseware产品上用户容量的需求。她也同意了客户只要求

应用程序同时支持20个用户实时聊天，这也变成了Proseware这款软件产品最重要的非功能性需求。

在团队动手写代码前，Petra组织了一个团队会议，以便所有团队成员都能参与项目讨论，从而尽力形成一个最初的用户故事积压工作。准备好故事后，团队就可以开始工作，以便在单个冲刺内产出一些可以演示的东西。

10.2 最初的产品积压工作

客户给Petra和团队提供了一份Proseware的特性清单。这个清单描述得很平淡，团队会议的目标就是将这个描述转换为一个或多个用户故事。

> 我们把Proseware看作让人们能够在线聊天的主站点。尽管如此，我们明白罗马不是一天就可以建成的，所以我们将目标做了一定程度的限制，以便尽可能快地看到一些可以交付的东西。
>
> 尽管我们想要允许用户相互发送图片或文件，但最关键的功能仍然以文字交流为基础。同一房间内的任何人都应该能接收到该房间内其他人发送的任何消息。
>
> 因为文字聊天非常重要，所以我们需要房间成员能够相互发送诸如HTML之类的格式化文本。当然，他们不能使用大量的无意义的图片和视频来破坏当前的聊天。
>
> Proseware中的某些会话应该不能被一些用户编辑，还有，一些用户应该只能看聊天信息，但不可以发送。

Petra把这个不长的描述带到了会议上，整个团队就开始尝试从描述中识别出用户故事。

10.2.1 从描述中挖掘故事

如上节所示，有时客户会平铺直叙地描述他们的期望。基于用户的描述，团队必须从中提出并商定好要实现的用户故事。Trey Research团队专门安排了一个会议来从客户提供的Proseware软件的描述中找出用户故事。

> PETRA：好了，现在开始会议。大家都有读过这个描述吗？（每个人都点了头，脸上露出一些热切的神情。）
>
> STEVE：这个描述里的东西很多啊！
>
> DAVID：不是很多啊，我们很可能一个冲刺就能搞定它们。（Steve咧嘴笑了起来。）
>
> DIANNE：我们首先能做的就是从这个描述中找出动词和名词：发送、接收、格式化、发送垃圾信息……房间、会话、用户、成员、聊天……
>
> STEVE：对的，描述里的动名词很多啊。
>
> PETRA：作为一个用户，我想要……以便……
>
> DIANNE：他们就只是用户吗？
>
> STEVE：看起来不是只有用户这一个角色。
>
> DAVID：他们一会叫他们用户，一会又叫他们成员。
>
> TOM：然而，你不是一个会话的参与者？

STEVE：也许吧，但是我们应该使用客户普遍使用的语言。还有一些围绕会话的角色只有读取权限，确实如此吗？

DIANNE：是的，听起来他们要说的是某种权限系统。

PETRA：请大家先提取出一个故事，好吧？

DAVID：作为一个成员，我想发送格式化的文本以便其他人能够……查看……是这些吗？

STEVE：大家都应该先集中在角色和行为上，对吧？Petra，你能给出业务价值信息吗？

PETRA：当然可以。

DIANNE：好的，但是David说的还是太大了，像是史诗，而不是用户故事。

PETRA：对不起，啥是史诗啊？

DAVID：史诗实际上就是个大型故事。Dianne认为这个故事太大，无法放进单个冲刺中，但是我不知道如何才能把它变小。

DIANNE：好的，什么比格式化文本更简单？先不要管HTML，这只是客户要求的一种实现而已！就当成任意类型的格式化文本，那么，什么会更简单呢？

DAVID：呃……非格式化的文本？

STEVE：也叫作纯文本。（David不好意思地笑了笑。）

DIANNE：作为一个房间成员，我想要给房间的其他人发送纯文本。（Petra在看有没有人反对，但是大家好像都默认达成共识了。）

TOM：漂亮，伙计们，我们的第一个用户故事产生了！（除了Petra，大家开始鼓掌。）

PETRA：但是客户要的是格式化代码啊？

STEVE：别担心，那是另外一个故事，会在纯文本故事交付后处理它。

PETRA：好的，作为一个房间成员，我想要给房间的其他人发送格式化文本。

STEVE：两个故事了，还有其他的吗？

DAVID：我想要创建房间，这个应该也是一个故事吧？

DIANNE：是的，我认为这一定会让那个人变成房间的主人，不是吗？作为房间的主人，我想要创建多个房间以对会话进行分类。

STEVE：好的。

TOM：客户也说了他们想要发送图片和文件。

PETRA：是的，那是另外一个故事。

STEVE：还有一个：我想创建只读的会话。

DIANNE：这已经是客户描述中的最后一句话了，所以我觉得我们已经找出了所有故事。

10.2.2　故事点估算

到现在为止，团队成员都同意他们从客户需求中找出了所有的用户故事。下面列出团队创建的用户故事卡片。

- ❑ 我想给房间其他人发送纯文本。
- ❑ 我想创建多个房间以对会话进行分类。
- ❑ 我想给房间其他人发送格式化文本。
- ❑ 我想发送图片和/或文件。
- ❑ 我想创建只读的会话。

团队把这些卡片都摆在了桌面上，他们开始准备做故事点估算。整个项目只有五个故事，所以团队选择使用计划扑克来估算需要的工作量。

PETRA：发送纯文本应该不会很难，对吧？

DAVID：我只需要一个小时！（Steve和Dianne睁大了眼睛，Tom转了转自己的眼珠。）

STEVE：啊，你不可以先做这个故事。因为没有房间，你都没有参与者。

DIANNE：是的，虽然这个故事是有依赖的，但是它并不会改变这个故事的估算。（Steve点头表示同意。）

PETRA：每个人都准备好点数了吗？

过了一会，所有团队成员都亮出了自己手中的计划扑克，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
3	3	5	5	1

PETRA：呃，打分有点乱。David，为什么只给一个点的估算啊？

DAVID：是啊，因为几乎没什么做的啊：只需要接受一些输入，然后把它们显示在屏幕上。

DIANNE：不只是这些，David。有很多事情你没有考虑到。我们需要把文本保存在某些地方，然后还可以读取出来。房间的其他成员也需要读取它。

STEVE：的确是这样的，Dianne是对的。我们需要在这里考虑好架构。

DAVID：噢，是哦。我原以为把输入显示在浏览器窗口上就可以了。当然，这样做的话，就只有写这个消息的人能看到，并不能共享给其他人。哇噢，的确比我刚才想的难得多！

PETRA：好的，那我们再来一次估算？

这一次所有团队成员很快就亮出了他们手中的计划扑克，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
5	5	5	5	5

STEVE：我们能把这个故事分割成“读”和“写”两个部分吗？

DIANNE：不好意思，你是指？

TOM：我想他的意思是我们可以有一个故事用来查看发送到一个房间中的所有消息，另外一个故事用来向指定房间发送消息。对吧，Steve？

STEVE：是的，我只是认为这样划分有助于我们从点数尽可能小的故事开始。我知道五个点的估算也不大，但是在开发的初始阶段，五个点还是挺大的。

DIANNE：好的，我也觉得这样划分会更好。那我们重新给发送消息估算点数吧？

PETRA	TOM	STEVE	DIANNE	DAVID
3	3	3	3	3

DIANNE：现在我们开始估算故事：查看已经发送到一个房间的消息。

PETRA	TOM	STEVE	DIANNE	DAVID
2	3	2	2	1

STEVE：Tom、David，你们同意两个点吗?

TOM：好的，两个点也可以。

DAVID：我也同意，两个点没问题。

STEVE：很好，它就是两个点的故事了。（Steve给这个故事标上了两个点，然后读出下一个故事。）我想创建多个房间以对会话进行分类。

DIANNE：我们该怎样演示这个呢?

TOM：Petra和我聊过这个，其中一个验收标准就是应该能够查看已经创建的房间列表。

DIANNE：这个也可以分割为“读”和“写”两个故事吧?

STEVE：嗯，我也这样想。

DIANNE：我们先给出读取部分的估算：我想要查看房间列表。

所有团队成员给出了他们的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
2	2	2	2	2

DIANNE：哇哦，心有灵犀啊。那么创建新房间的故事呢?

所有团队成员再次给出了他们的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
2	2	2	2	1

STEVE：David，你觉得两个点可以吗?

DAVID：是的，我可以接受。

STEVE：下一个故事：我想要给房间的其他成员发送格式化文本。

PETRA：这只是在前面故事的基础上增加个格式化文本的工作，对吧?

STEVE：是的，前提是纯文本的故事必须先完成，因为它是这个故事的前提条件。

TOM：客户到底需要什么样的格式化呢? 内嵌的图片或其他啥东西?

DAVID：图片是另外一个故事。我认为这里的格式化是指诸如粗体、斜体、下划线等。

PETRA：是的，这也是客户想要的。HTML是一个他们能想到的格式，但是我不认为我们想要做HTML格式。只是简单的文本格式化，其他都不需要。

STEVE：好的，准备好估算了嘛?

所有团队成员给出了他们的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
1	1	1	1	1

STEVE：啊哈，我们很有默契啊。那么，下一个故事：我想要发送图片和/或文件。

DIANNE：我不喜欢故事里的“和/或”。我们能把这种歧义消除掉吗？

TOM：好的，那么发送图片和发送文件之间有区别吗？

PETRA：根据我和客户的讨论，他们是想用户能在浏览器中直接查看图片，而文件则是要下载到用户的机器上。

DAVID：这实际上是两个故事了，不是吗？一个是下载文件，另外一个是显示图片。

DIANNE：完全正确，作为一个故事有些太大了，我们是应该把它们分开。而且似乎下载文件这个故事会更小些，因此我们应该先完成它，后面显示图片的故事会以下载文件为基础。

STEVE：我同意。那我们先就把它分成两个故事吧。我想要给房间的其他成员发文件。我想在房间显示图片。我们来一个一个估算。

所有团队成员给出了第一个故事的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
5	5	3	3	3

STEVE：哦？你们两个为何给的是五个点的估算？

TOM：测试这个故事会比较难。我们还需要考虑很多不常见的测试用例。比如，如果发送的文件很大？如果文件是病毒？是否要在我们的服务器上保持文件？如果要的话，要保存多久？

STEVE：我同意你的观点。我刚才只考虑了开发的工作量，抱歉。我现在也相信你需要花费很多时间来设计测试用例。

DAVID：我们是要重新估算，还是按照五个点的估算来？

STEVE：我觉得五个点是合适的。

DIANNE：我也一样。

STEVE：那么在浏览器显示图片呢？

PETRA：这只是在网页上给用户展示下载的图片，对吧？

STEVE：是的，上传文件的功能已经包括在上一个故事里面了，所以这个故事只是在网页上显示文件。

团队成员给出了这个故事的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
3	3	3	3	5

STEVE：David，这个还需要五个点吗？

10

DAVID：的确是的。我认为这个故事实际要比它看起来难一些。如果图片不适宜显示呢？如果用户很多，而且图片下载时间比较长呢？这会给我们的服务器带来较大的负载。

DIANNE：我也同意这些都是要担心的事情，但是我个人认为它们暂时超出了需求范围。Petra，你要记得去问问客户是否想要内容过滤的功能，不仅仅针对图片，也可以用于文字。但是，现在我们只需要支持最多20个用户，这应该不会给服务器带来明显的负载。

PETRA：内容过滤功能这个主意很好，David。

STEVE：David，现在可以接受三个点的估算吗？

DAVID：如果是这样，我认为实际上两个点就足够了！不过，三个点也可以。

STEVE：不错，现在只剩最后一个故事了：我想要创建只读的会话。

所有团队成员给出了他们的点数估算，如下所列。

PETRA	TOM	STEVE	DIANNE	DAVID
8	1	5	8	3

STEVE：天哪，这次打分有些随意啊！Tom，你的一个点的依据是什么？

TOM：抱歉，我没考虑到分析和开发的工作量了。只是这个故事真的要比其他故事容易测试得多。

STEVE：好的。Petra、Dianne，你们两个的八点又是怎么得出来的？

DIANNE：到现在为止，还没有涉及必要的角色或权限。我认为这需要很多工作要做。

DAVID：噢，我不认为我们现在需要角色或权限的功能。我认为我们只需要把某个会话标记为只读即可。

STEVE：只读是针对每个用户而言的。但是，我依然不认为我们需要复杂的权限或角色基础架构。我认为应该先从最简洁的方案开始实现，后期再做更完善的设计。

DIANNE：这样也行。

PETRA：好，我相信大家的选择。

STEVE：平均估算值为五个点，我们就按五个点来？（大家都表示同意。）

STEVE：David，你现在还认为这些工作一个冲刺就做得完吗？

DAVID：嘿嘿……不。

会议圆满结束。

10.3　总结

整个会议，团队先对客户提供的需求描述进行了分解，并得到了若干个用户故事。然后，整个团队逐个对故事点数进行了估算，并创建了一个排好序的积压工作。他们现在可以动工开发了。

注意，讨论过程中是允许开发过程中所有角色参与估算的，包括分析、实现和测试。对于每个故事，团队对完成整个故事所需的工作量都达成了共识。

第 11 章

自适应实例冲刺1

完成本章学习之后，你将学到以下技能。

- 观摩团队第一个冲刺中计划会议的过程。
- 追踪第一个用户故事的实现和演进过程。
- 观摩团队第一个冲刺中冲刺演示和回顾会议的过程。

本章中，Trey Resarch团队实现了前四个用户故事。这四个故事是项目第一个冲刺的交付目标。团队还设置了下面的冲刺目标。

> 能展示动态的房间列表，能创建新房间，能查看在房间发布的信息，同一房间内至少两个用户可以共享消息。

通过定义冲刺目标，团队设定了一些有挑战但依然可以在这个冲刺完成的任务项。所有具体的工作项都必须符合冲刺目标，以确保团队的工作没有偏离客户的需求。

11.1 计划会议

团队安排了冲刺计划会议并且带去了与冲刺目标相关的用户故事。对于第一个冲刺，要实现的故事包括以下这些。

- 我想要创建多个房间以对会话进行分类。
- 我想要查看代表会话的房间的列表。
- 我想要查看发送到一个房间内的消息。
- 我想要给房间内的其他成员发送纯文本消息。

团队开始了他们的讨论过程。

> PETRA：好，开始开会了。今天只要讨论四个故事，但它们都需要在这个冲刺交付，因此我希望今天的会议能做出一些关键的决定。
>
> STEVE：嗯，可能吧。我们肯定需要在开始前考虑一些技术问题。大家都有啥建议？
>
> DIANNE：这是一个网络应用程序，所以我们应该使用我们都觉得熟悉和满意的平台。似乎ASP.NET很明显是一个很好的选择，大家都同意吗？
>
> STEVE：我同意。
>
> DAVID：呃。说实话，我认为这是一个能完美应用Node.js的应用程序。

TOM：不好意思，我没有使用过Node。有人用过吗？

STEVE：我也不是很熟悉。我认为Dianne是对的，我们应该坚持使用我们都知道的方式。

DIANNE：基于MVC，我们已经有了很多基础的代码结构可供我们使用。大家一起看看这个时序图。

Dianne向团队展示了UML时序图，如图11-1所示。

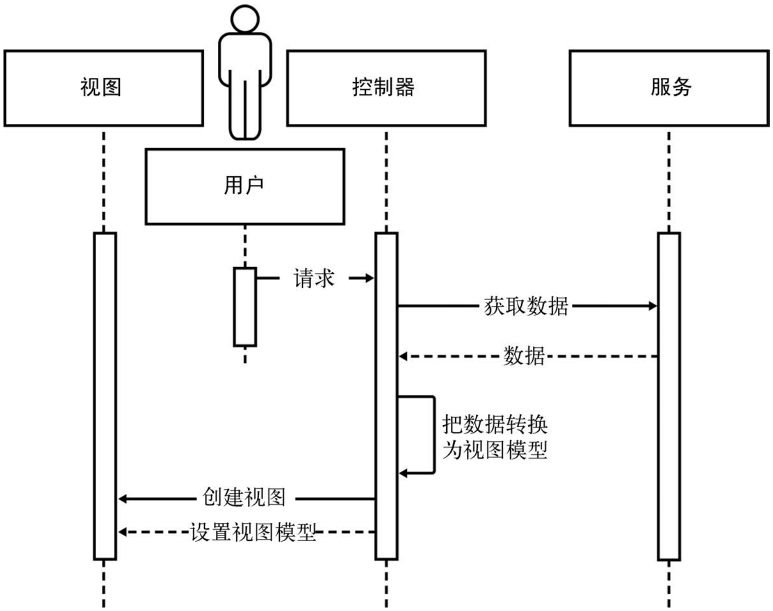


图11-1　Dianne的时序图能描述通用的MVC应用程序结构

STEVE：好的，我理解这个时序图，目前也同意你的设计。

DAVID：但是，这个时序图看起来和我们要讨论的房间和消息没有任何关系啊。

DIANNE：表面看是没有关系，不过我会给出进一步的解释的。这是一个通用的时序图。我决定不去为每个用户故事创建时序图，而是创建一个通用的方案以便让我们都清楚需要什么样的类，至少是那些目前看起来就需要的类。

DAVID：噢，我明白了。所以获取数据（Get data）就是获取房间（Get room）和获取消息（Get message）的占位符，对吗？

DIANNE：是的，你的理解是对的。

STEVE：那么，服务部分是指什么呢？它是事件驱动的消息系统或者是某种我们要操作的数据库系统吗？

DIANNE：HTTP是一个离线的无状态协议。我们必须为消息引入服务。

DAVID：但是最后一次检查的只有消息。我们并不想要总是返回所有消息。

STEVE：是的，这样做扩展性也不太好。

DIANNE：说到这里，这是一个可扩展的方案吗?

PETRA：问得好。尽管用户短期只要求最多支持20个用户，但可以说这个需求很快就会改变的。而且我们现在也可能知道最大限制数目。我们不应该限制应用程序后续的扩展性。

STEVE：后续并没有限制扩展性。只要我们从一开始就编写具备良好自适应能力的代码，就应该能够在我们取得进展的过程中发现最合适的架构方案。

TOM：那么用户界面呢？如果我们打算下一周就做演示，它就应该有些演示的模样了。

PETRA：Tom是对的。很多时候，演示功能虽然很强大，但是用户界面却很糟糕，这会让用户很大程度上失去对我们的信心。

DAVID：好了，MVC使用引导程序，所以我们可以实现一些引导模板。我们会需要一个房间列表页面和一个房间内消息的页面。即使是很简单的风格，但至少要将它们设计得现代感强一些。

PETRA：不要担心风格过于简单。我认为后续无论如何都要允许用户自定义显示主题的。在早期阶段，任何试图对用户界面做额外美化的努力都可能是白费功夫。

STEVE：Tom，有关测试，你有什么问题吗?

TOM：没有，根据我们的冲刺目标，我确认最多只有两个用户，所以我可以把自动加载测试朝后推迟一些。在我把自动化测试环境搭建起来之前，可以先集中做一些手动测试。当然，我肯定希望你们三个人都自觉完成单元测试，所以，如果后续你们需要有关测试的帮助，要记得及时告诉我。

STEVE：很好，我们就先这样。可以在需要时再进一步讨论具体的细节问题。

会议结束后，团队已经开始准备实现第一个故事了。

11.2 “我想创建多个房间以对会话进行分类”

两天后，David说他做好创建房间故事代码评审的准备了。Dianne直接去了David的开发座位前，两个人一起讨论代码。讨论的目的是为了识别实现的优缺点。这样的同行评审能给开发人员一个机会来识别是否有些实现没有达标或缺乏适应变更的能力，因此在将代码提交到源代码控制系统前，评审是最后一次变更方案实现的机会。

11.2.1 控制器

当David开始实现控制器时，他向Dianne征求意见。

DAVID：现在，我已经按照你的建议应用了MVC模式。控制器本身并没做多少事情，它只是委托`IRoomRepository`接口来和我们要使用的持久存储模块交互。

DIANNE：让我们先看看控制器的代码吧。

David给Dianne展示了控制器的实现代码，如代码清单11-1所示。

代码清单11-1　RoomController是Create请求的入口点

```
public class RoomController : Controller
{
    private readonly IRoomRepository roomRepository;
    public RoomController(IRoomRepository roomRepository)
    {
        Contract.Requires<ArgumentNullException>(roomRepository != null);

        this.roomRepository = roomRepository;
    }

    [HttpGet]
    public ActionResult List()
    {
        return View();
    }

    [HttpGet]
    public ActionResult Create()
    {
        return View(new CreateRoomViewModel());
    }

    [HttpPost]
    public ActionResult Create(CreateRoomViewModel model)
    {
        ActionResult result;

        if(ModelState.IsValid)
        {
            roomRepository.CreateRoom(model.NewRoomName);

            result = RedirectToAction("List");
        }
        else
        {
            result = View("Create", model);
        }

        return result;
    }
}
```

DIANNE：好的，看起来不错。我们从构造函数得到了注入的IRoomRepository接口依赖，这样就给了我们一些灵活性。而且通过构造函数入口的契约能保证它不为空，这很好。

DAVID：是的，我也有单元测试来确保这个不为空的前置条件。如果你要看，我可以给你看看。

DIANNE：那样很好，但是现在你先能为我解释一下POST请求处理器的Create方法吗？具体讲，为什么该方法在委托存储库的CreateRoom方法后还要重定向到列表动

作上呢？

DAVID：这是PRG（Post-Redirect-Get）模式。从根本上讲，如果我们这个时候直接返回List视图的话，用户任何刷新页面的尝试都会产生第二个POST请求。这将意味着我们会尝试用相同的名称再创建一个房间，而且从用户体验角度上讲，这肯定是不对的。

DIANNE：漂亮！这个模式现在用在这里很合适。

DAVID：有件事我还不太确定，是直接实例化新的CreateRoomViewModel，还是应该使用一个工厂类？

DIANNE：问得好。我个人认为在这个用例中是不需要使用工厂的。因为看起来返回值的类型不太可能发生变化。视图模型的应用场景也比较有限，所以不需要增加中间层来委托工厂返回视图模型的实例。

控制器单元测试

David打开了包括单元测试的文件，请求Dianne对他的工作进行同行评审。文件内容如代码清单11-2所示。

DAVID：以下是RoomController构造函数的单元测试。

代码清单11-2　通过单元测试验证RoomController构造函数

```
[Test]
public void ConstructingWithoutRepositoryThrowsArgumentNullException()
{
    Assert.Throws<ArgumentNullException>(() => new RoomController(null));
}

[Test]
public void ConstructingWithValidParametersDoesNotThrowException()
{
    Assert.DoesNotThrow(() => CreateController());
}
```

DAVID：第一个测试确保了RoomController类的IRoomRepository接口字段始终有效。

DIANNE：是的，这是RoomController类的一个隐含的契约，那就是存储库字段必须不为空且有效。

DAVID：接下来的两个测试是为了对响应GET请求的Create方法行为的断言。

David下拉文件滚动条并找到了如代码清单11-3所示的两个测试。

代码清单11-3　针对Create动作的GET请求的单元测试


```
[Test]
public void GetCreateRendersView()
{
    var controller = CreateController();

    var result = controller.Create();
```

```
    Assert.That(result, Is.InstanceOf<ViewResult>());
}

[Test]
public void GetCreateSetsViewModel()
{
    var controller = CreateController();

    var viewResult = controller.Create() as ViewResult;

    Assert.That(viewResult.Model, Is.InstanceOf<CreateRoomViewModel>());
}
```

DAVID：我们需要明白两件事：请求会返回一个ViewResult类实例，以及Mode属性的期望类应该是CreateRoomViewModel类。

DIANNE：你能把这两个单元测试合并为同时有两个断言的一个测试吗？

DAVID：我觉得可以，但是合并起来后的单元测试名称就不会像现在这样清晰地表达它的断言意图了。我更愿意编写粒度合适的测试，这些目标明确的测试包含了尽可能少的断言，也有助于更容易地识别错误行为。

DIANNE：挺好的。那么如何断言POST请求中那个委托服务完成动作的Create动作呢？

DAVID：这是针对它的单元测试。

David再次向下滚屏，给Dianne展示了如代码清单11-4中所示的单元测试。

代码清单11-4　针对Create动作的POST请求的单元测试

```
[Test]
[TestCase(null)]
[TestCase("")]
[TestCase("    ")]
public void PostCreateNewRoomWithInvalidRoomNameCausesValidationError(string roomName)
{
    var controller = CreateController();

    var viewModel = new CreateRoomViewModel { NewRoomName = roomName };

    var context = new ValidationContext(viewModel, serviceProvider: null, items: null);
    var results = new List<ValidationResult>();

    var isValid = Validator.TryValidateObject(viewModel, context, results);

    Assert.That(isValid, Is.False);
}

[Test]
[TestCase(null)]
[TestCase("")]
[TestCase("    ")]
```

```
public void PostCreateNewRoomWithInvalidRoomNameShowsCreateView(string roomName)
{
    var controller = CreateController();

    var viewModel = new CreateRoomViewModel { NewRoomName = roomName };
    controller.ViewData.ModelState.AddModelError("Room Name", "Room name is required");
    var result = controller.Create(viewModel);

    Assert.That(result, Is.InstanceOf<ViewResult>());

    var viewResult = result as ViewResult;
    Assert.That(viewResult.View, Is.Null);
    Assert.That(viewResult.Model, Is.EqualTo(viewModel));
}

[Test]
public void PostCreateNewRoomRedirectsToViewResult()
{
    var controller = CreateController();

    var viewModel = new CreateRoomViewModel { NewRoomName = "Test Room" };
    var result = controller.Create(viewModel);

    Assert.That(result, Is.InstanceOf<RedirectToRouteResult>());

    var redirectResult = result as RedirectToRouteResult;
    Assert.That(redirectResult.RouteValues["Action"], Is.EqualTo("List"));
}

[Test]
public void PostCreateNewRoomDelegatesToRoomRepository()
{
    var controller = CreateController();

    var viewModel = new CreateRoomViewModel { NewRoomName = "Test Room" };
    controller.Create(viewModel);

    mockRoomRepository.Verify(repository => repository.CreateRoom("Test Room"));
}
```

DAVID：前两个测试对必须的房间名称字段进行了断言。为了创建房间，必须要提供房间名称。如果验证出错，用户会被拉回创建房间的页面。

DIANNE：用TestCase属性来为单元测试指定错误的房间名称，这种方式不错，我很喜欢。

DAVID：谢谢夸奖。我认为这样可以少写点测试方法。

DIANNE：到目前都挺好，我们看下一个吧。

11.2.2 房间存储库

David不太确定自己对房间存储库的实现是否正确。他打开该类的实现文件以便Dianne能提

供评审建议和意见。文件内容如代码清单11-5所示。

DAVID：我创建了IRoomRepository接口的ADO.NET版本的实现。你看看。

代码清单11-5　AdoNetRoomRepository允许在任何ADO.NET兼容数据库中存储Room数据

```
public class AdoNetRoomRepository : IRoomRepository
{
    public AdoNetRoomRepository(IConnectionIsolationFactory factory)
    {
        this.factory = factory;
    }

    public void CreateRoom(string name)
    {
        factory.With(connection =>
        {
            using(var transaction = connection.BeginTransaction())
            {
                var command = connection.CreateCommand();
                command.CommandText = "dbo.create_room";
                command.CommandType = CommandType.StoredProcedure;
                command.Transaction = transaction;
                var parameter = command.CreateParameter();
                parameter.DbType = DbType.String;
                parameter.ParameterName = "name";
                parameter.Value = name;
                command.Parameters.Add(parameter);

                command.ExecuteNonQuery();
            }
        });
    }

    private readonly IConnectionIsolationFactory factory;
}
```

DAVID：我在这里使用了工厂隔离模式，我委托一个工厂接口来控制数据库连接对象的生命周期。CreateRoom方法体只是一些标准的ADO.NET代码。

DIANNE：嗯，但是我也不太确定这里应用工厂隔离模式是否合适。我去跟Steve确认一下。

1. 工厂隔离模式的误用

Dianne叫了Steve过来一起看代码以确认David工厂隔离模式的使用是否正确。

STEVE：有什么事吗？

DIANNE：帮我看看这个类的实现。David在这里使用了工厂隔离模式，但是我觉得可能不太合适。

STEVE：嗯，的确不合适，我知道问题是什么。David，你用的是ADO.NET，对吧？我们看看你的工厂类的实现代码。

David打开了文件。Dianne和Steve看到的源代码如代码清单11-6所示。

代码清单11-6 使用工厂隔离的目标是管理数据库连接的生命周期

```
public class AdoNetConnectionIsolationFactory : IConnectionIsolationFactory
{
    private readonly IApplicationSettings applicationSettings;
    private readonly DbProviderFactory dbProviderFactory;
    public AdoNetConnectionIsolationFactory(IApplicationSettings applicationSettings)
    {
        this.applicationSettings = applicationSettings;
        this.dbProviderFactory = DbProviderFactories
                .GetFactory(applicationSettings.GetValue("DatabaseProviderName"));
    }

    public void With(Action<IDbConnection> action)
    {
        using(var connection = dbProviderFactory.CreateConnection())
        {
            connection.ConnectionString = applicationSettings
                    .GetValue("ProsewareConnectionString");
            connection.Open();

            action(connection);
        }
    }

}
```

STEVE：DbProviderFactory类的CreateConnection方法返回的是IDbConnection对象。

DAVID：嗯，是的。

DIANNE：我明白问题所在了。工厂隔离模式只适合在工厂生成对象不确定实现IDisposable接口时使用。而IDbConnection接口已经实现了IDisposable接口，所以它能保证所有自己的实现类都实现了公共的Dispose方法。

STEVE：的确。所以你根本不需要使用工厂隔离，因为……

DAVID：因为我只需要换成使用using语句块就可以了。

DIANNE：对。

DAVID：好的，所以这应该只是一个普通的工厂？

DIANNE：我不这么认为。你看这个类的具体实现，已经很好地对客户端隐藏了DbProviderFactory类。所以我认为这里的工厂中间层是没必要的。

STEVE：我同意。

DAVID：但是这就意味着我要在房间存储库的实现中直接调用DbProviderFactory类。这样做不够灵活，尤其是DbProviderFactory类还是静态的。

STEVE：是的，所以DbProviderFactory类最好不要是静态的。但是房间存储库的实现直接依赖ADO.NET，所以静态的DbProviderFactory也不算大问题。引入工厂

11

会让本来只在存储库实现中操心的细节变得公开了。

DAVID：好的，有道理。那么我们可以直接忽略对这个静态类的依赖，因为它太通用了，对吧？

DIANNE：是的。你可以考虑为DbProviderFactory类引入接口，然后为其增加有意义的修饰或适配。任何可替换的行为都应该从更高层次的IRoomRepository接口实现中注入。

DAVID：我明白了。让我重构一下，然后再让你们看看。

2. 重构

David对房间存储库进行了重构，重构的结果如代码清单11-7所示。他再把Dianne和Steve叫来以听取他们的建议和意见。

代码清单11-7　重构后的房间存储库直接使用了DbProviderFactory

```
public class AdoNetRoomRepository : IRoomRepository
{
    private readonly IApplicationSettings applicationSettings;
    private readonly DbProviderFactory databaseFactory;
    public AdoNetRoomRepository(IApplicationSettings applicationSettings,
  DbProviderFactory databaseFactory)
    {
        Contract.Requires<ArgumentNullException>(applicationSettings != null);
        Contract.Requires<ArgumentNullException>(databaseFactory != null);

        this.applicationSettings = applicationSettings;
        this.databaseFactory = databaseFactory;
    }

    public void CreateRoom(string name)
    {
        using(var connection = databaseFactory.CreateConnection())
        {
            connection.ConnectionString =
  applicationSettings.GetValue("ProsewareConnectionString");
            connection.Open();

            using(var transaction = connection.BeginTransaction())
            {

                var command = connection.CreateCommand();
                command.CommandText = "dbo.create_room";
                command.CommandType = CommandType.StoredProcedure;
                command.Transaction = transaction;
                var parameter = command.CreateParameter();
                parameter.DbType = DbType.String;
                parameter.ParameterName = "name";
                parameter.Value = name;
                command.Parameters.Add(parameter);

                command.ExecuteNonQuery();
            }
```

```
        }
    }

}
```

DAVID：我做了些改动。我把原来连接工厂类的代码继承到存储库类中。我也不想随意使用静态的DbProviderFactories类。DbProviderFactory类至少应该是能够提供扩展点的抽象类。这样做，我们就可以使用场景的依赖注入方式，只是注入的是抽象类，而不是接口。

DIANNE：很好，我觉得你改得很好。

STEVE：你能给我看看注册配置DbProviderFactory类的控制反转容器吗？我想知道你的注册过程。

David给Steve和Dianne展示了使用Unity的注册代码，如代码清单11-8所示。

代码清单11-8 DbProviderFactory的统一控制反转（IoC）注册

```
container.RegisterType<DbProviderFactory>(new InjectionFactory(c =>
  DbProviderFactories.GetFactory(
  c.Resolve<IApplicationSettings>().GetValue("DatabaseProviderName"))));
```

STEVE：挺好。我原以为因为涉及静态类会让注册显得不自然，但我认为你做得很好，已经把它从存储库的实现中抽象出来了。

DIANNE：我觉得可以提交代码了，你认为呢，Steve?

STEVE：嗯，提交吧。这样我们第一个故事的编码工作就完成了。

David提交了代码后，团队就开始了下一个故事的开发。

11.3 “我想查看代表会话的房间的列表”

第二天，David完成了第二个故事，可以查看所有已经创建房间的列表。在提交代码前，他又叫了Dianne来做评审。

DAVID：我修改了RoomController，这样它就可以读取房间列表并在一个页面上显示房间列表了。

DIANNE：好的，我们就从发生变更的地方开始。给我看看你对RoomController的修改。

David给Dianne展示的代码如代码清单11-9所示。为了让代码片段更好地表达当前讲解的重点，下面示例中略去了房间的创建动作代码。

代码清单11-9 RoomController有一个用于列出房间的新动作

```
public class RoomController : Controller
{
    private readonly IRoomRepository roomRepository;
    private readonly IRoomViewModelMapper viewModelMapper;
```

```
    public RoomController(IRoomRepository roomRepository, IRoomViewModelMapper mapper)
    {
        Contract.Requires<ArgumentNullException>(roomRepository != null);
        Contract.Requires<ArgumentNullException>(mapper != null);

        this.roomRepository = roomRepository;
        this.viewModelMapper = mapper;
    }

    [HttpGet]
    public ActionResult List()
    {
        var roomListViewModel = new RoomListViewModel();

        var allRoomRecords = roomRepository.GetAllRooms();

        foreach(var roomRecord in allRoomRecords)
        {
            roomListViewModel.Rooms
                    .Add(viewModelMapper.MapRoomRecordToRoomViewModel(roomRecord));
        }

        return View(roomListViewModel);
    }

}
```

DAVID：这里引入了一个新的构造函数参数：一个映射器，也相应地增加了代码契约来保证它不为空，另外也为它编写了对应的单元测试。

DIANNE：很好。但我觉得还可以对一个地方进行改动。跟我讲一遍你的List方法，我会让你知道我们还可以简化这个类。

DAVID：List方法实际上有两个部分。第一，房间存储库会查询以获取所有房间记录。但是，这些房间记录只是数据模型，不是视图模型，所以我引入了映射器来完成类间的转换。这个工作会委托给映射器接口。在房间模型数据转换为RoomViewModel类的对象后，我们就可以把它们传递给视图以显示房间列表。

DIANNE：我认为你的抽象有个漏洞。IRoomRepository接口返回的RoomRecord类属于更下面的数据存储层。我不确定是否应该把它直接抛出到控制器中。你引入映射器的目的正是为了把数据层类型转换为视图数据类型。

DAVID：但是我们还有其他的方式吗？我只能得到RoomRecord实例，但是我也需要返回RoomViewModel类的对象。

DIANNE：嗯。但是控制器不应该知道RoomRecord类的存在，也不应该知道类的映射过程。让控制器既不依赖IRoomRepository接口也不依赖IRoomViewModelMapper接口，而是依赖一个新的接口，怎么样？我们可以把这个新接口命名为IRoomViewModelService。它可以让控制器不知道RoomRecord的存在。这个新服务的实现能够使用房间存储库和映射器来返回控制唯一了解的RoomViewModel类的实例。

DAVID：我明白。但是，IRoomRepository接口中已有的CreateRoom方法调用怎么办？没有存储库，控制器没办法调用这个方法。

DIANNE：问得好。这里我建议你做接口分离。可以把IRoomViewModelService接口分离为IRoomViewModelReader和IRoomViewModelWriter两个接口。这样可以给我们以后变更实现的读和写都提供独立的扩展点。

DAVID：好的，这样分离后，我的单元测试代码也要做很多改动。看来只有等重构完后再请教你了。

DIANNE：这会影响我们的冲刺目标的完成吗？需要为交付期限做折中方案吗？

DAVID：不需要，重构工作量不大。我应该能在一两个小时内搞定。

DIANNE：很好。

David开始修复Dianne指出的问题。

重构

David花了几个小时完成了Dianne建议的改动。然后他叫Dianne对重构后的代码再做一次检查。

DAVID：这是我重构后的控制器。你看看还有什么问题吗？

代码清单11-10展示了重构后的新控制器。

代码清单11-10 控制器根据读取和写入接口进行了重构

```
public class RoomController : Controller
{
    private readonly IRoomViewModelReader reader;
    private readonly IRoomViewModelWriter writer;
    public RoomController(IRoomViewModelReader reader, IRoomViewModelWriter writer)
    {
        Contract.Requires<ArgumentNullException>(reader != null);
        Contract.Requires<ArgumentNullException>(writer != null);

        this.reader = reader;
        this.writer = writer;
    }

    [HttpGet]
    public ActionResult List()
    {
        var roomListViewModel = new RoomListViewModel(reader.GetAllRooms());

        return View(roomListViewModel);
    }

    [HttpGet]
    public ActionResult Create()
    {
        return View(new RoomViewModel());
    }

    [HttpPost]
    public ActionResult Create(RoomViewModel model)
```

```
    {
        ActionResult result;

        if(ModelState.IsValid)
        {
            writer.CreateRoom(model.Name);

            result = RedirectToAction("List");
        }
        else
        {
            result = View("Create", model);
        }

        return result;
    }

}
```

DIANNE：现在你已经完全剔除了对映射器的依赖。注入IRoomViewModelReader和IRoomViewModelWriter两个接口意味着我们后面能够分别改动读和写的实现，可以防止我们后面更改到CQRS架构。

DAVID：CQRS是什么来着，你能给个提示吗？

DIANNE：CQRS即命令查询责任分离。按照这种模式，你的应用程序中的命令和查询是非对称的。现在这个项目里，我们可能会向事务存储写入数据，但却从非事务型的文档存储中读取数据。我认为Steve已经在为当前架构不能很好适应扩展的情况准备可能的新架构了，我们都在这样猜想。

DAVID：我记起来了，命令查询责任分离听起来对于我们的场景挺合适的。那我们开始看看IRoomViewModelReader和IRoomViewModelWriter这两个接口的实现吧？

Dianne点点头，David打开包含实现类的文件，内容如代码清单11-11所示。

代码清单11-11　一个更接近于原始控制器代码的服务实现

```
public class RepositoryRoomViewModelService : IRoomViewModelReader, IRoomViewModelWriter
{
    private readonly IRoomRepository repository;
    private readonly IRoomViewModelMapper mapper;
    public RepositoryRoomViewModelService(IRoomRepository repository,
  IRoomViewModelMapper mapper)
    {
        Contract.Requires<ArgumentNullException>(repository != null);
        Contract.Requires<ArgumentNullException>(mapper != null);

        this.repository = repository;
        this.mapper = mapper;
    }

    public IEnumerable<RoomViewModel> GetAllRooms()
    {
        var allRooms = new List<RoomViewModel>();
```

```
        var allRoomRecords = repository.GetAllRooms();
        foreach(var roomRecord in allRoomRecords)
        {
            allRooms.Add(mapper.MapRoomRecordToRoomViewModel(roomRecord));
        }
        return allRooms;
    }

    public void CreateRoom(string roomName)
    {
        repository.CreateRoom(roomName);
    }

}
```

DAVID：我决定在一个类中同时实现两个接口，因为它们都依赖IRoomRepository接口。但是，我认为它和原有的控制器类实现很接近，那么我们这样重构后又得到什么好处了呢？

DAINNE：最大的好处是，现在的控制器的主要职责更加清晰了。它现在不再负责从记录到视图模型的转换，而是专心做校验。不再需要知道RoomRecord类的控制器比原来的好很多。

DAVID：是的，可是我认为是不是做得有点过了，你觉得呢？

DIANNE：它并没有真正违背单一职责原则。如果我们不使用存储库，那么我们也不需要映射器。这种变化会影响到架构，控制器不应该担心这样的改变。

然后，David和Dianne都同意可以提交这个故事的实现了。

11.4 “我想查看发送到一个房间内的消息”

通过结对编程，Dianne和David在一起实现下一个故事。在Dianne编写代码时，David在旁边阅读并提出建议。

> **注意** **结对编程**（pair programming）是名为**极限编程**（Extreme Programming，XP）的敏捷软件开发方法中一个很常见的实践活动，它是指两个开发人员一起完成同一个指定的功能。当结对一方编写代码时，另一方能考虑正在实现的方法、类或单元测试是否还有更好的方式。

DIANNE：我们从控制器开始吧？

DAVID：好的，从这里开始应该比较好。

DIANNE：现在我们需要为查看房间内的信息增加一个新的HttpGet处理器。我们叫它什么呢？

DAVID：就叫GetMessages吧，怎么样？

DIANNE：听起来很好，但ASP.NET MVC模式使用控制器名称和方法名称来构造请求的URL。

DAVID：是啊，我忘记这个了。那就叫作Messages吧？作为Roomntroller的一部分后，URL看起来会是/Room/Messages这样子。

DIANNE：好，这个名称可以。我们还需要一个参数来表示要查看哪个房间的消息。

DAVID：我们有房间标识，它能唯一表示一个房间。我们就用它如何？

DIANNE：是个整型还是长整型？

DAVID：我觉得整型就够了。

DIANNE：我们也需要为这个视图增加相应的视图模型。按照你现在一直在用的命名规则，就叫它MessageListViewModel，怎么样？

DAVID：可以。我们只要把MessageListViewModel类实例传给控制器的View方法就可以得到一个ViewResult类的返回。

DIANNE：IRoomViewModelReader会需要一个新的方法来查询房间内的消息。

DAVID：我觉得叫它GetRoomMessages就好，它也有个表示房间标识的参数。

DIANNE：我会先在接口中定义，然后在实现类中提供一个占位实现以便我们后面决定控制器怎么做改动。

Dianne创建的方法如代码清单11-12所示。

代码清单11-12　Messages方法按ID检索所有与房间相关联的消息

```
[HttpGet]
public ActionResult Messages(int roomID)
{
    var messageListViewModel = new MessageListViewModel(reader.GetRoomMessages(roomID));

    return View(messageListViewModel);
}
```

DIANNE：看起来不错。现在我们可以实现IRoomViewModelReader接口中的Get-RoomMessages方法了。

DAVID：好的，现在的实现只使用了IRoomRepository接口。我认为我们应该新增一个IMessageRepository接口以便能获取消息。

DIANNE：同意。我们只需要从构造函数注入它即可，就像其他依赖一样。

DAVID：当我们有了消息存储库后，我们需要请求它来根据房间标识获取消息，然后把消息传递给映射器。

DIANNE：那么这个新的映射器负责将存储库记录转换为视图模型，这样控制器才可以使用，对吧？现在看，好像只有一个IRoomViewModelMapper接口可用。貌似应该增加一个新的IMessageViewModelMapper接口了。我们能把这个接口命名定义得更加通用吗？

DAVID：IViewModelMapper这个名称怎么样？这个名称表明它可以将所有记录转换为相应的视图模型。

DIANNE：代码写成下面这样，你看怎么样？

Dianne给David展示的代码如代码清单11-13所示。

代码清单11-13 将记录映射到ViewModel的类现在包含一个用于检索房间消息的方法

```
public class RepositoryRoomViewModelService : IRoomViewModelReader, IRoomViewModelWriter
{
    private readonly IRoomRepository roomRepository;
    private readonly IMessageRepository messageRepository;
    private readonly IViewModelMapper mapper;
    public RepositoryRoomViewModelService(IRoomRepository roomRepository,
  IMessageRepository messageRepository, IViewModelMapper mapper)
    {
        Contract.Requires<ArgumentNullException>(roomRepository != null);
        Contract.Requires<ArgumentNullException>(messageRepository != null);
        Contract.Requires<ArgumentNullException>(mapper != null);

        this.roomRepository = roomRepository;
        this.messageRepository = messageRepository;
        this.mapper = mapper;
    }

    public IEnumerable<MessageViewModel> GetRoomMessages(int roomID)
    {
        var roomMessages = new List<MessageViewModel>();
        var roomMessageRecords = messageRepository.GetMessagesForRoomID(roomID);
        foreach(var messageRecord in roomMessageRecords)
        {
            roomMessages.Add(mapper.MapMessageRecordToMessageViewModel(messageRecord));
        }
        return roomMessages;
    }

}
```

DAVID：嗯，这样写可以。

David和Dianne然后接着实现了消息存储库的GetMessageForRoomID方法，该方法从Microsoft SQL Server数据库中加载消息。提交所有代码后，他们开始开发下一个用户故事。

11.5 “我想给房间内的其他成员发送纯文本消息”

实现最后一个故事时，David和Dianne交换了位置，David写代码，Dianne提建议。

DAVID：这个故事应该比较容易实现，因为我们已经有了现成的读写数据的模式。

DIANNE：一定程度上是的。但是记住，这个故事的一个需求是说所有发送消息的动作是并行发生的。

DAVID：噢。所以不能做全页完全回发。

DIANNE：嗯，不可以。用户界面会通过一个AJAX请求发送消息数据。

> **注意** 一些术语定义：AJAX（Asynchronous JavaScript and XML，AJAX）指异步的JavaScript和XML。XML（Extensible Markup Language，XML）是指可扩展的标记语言。AJAJ（Asynchronous JavaScript and JSON，AJAJ）是指异步的JavaScript和JSON。JSON（JavaScript Object Notation，JSON）是指JavaScript对象标记法。

11

DAVID：等等，你是说AJAJ，对吗？

DIANNE：我觉得我是要说AJAJ！

DAVID：这样能让控制器行为有何不同？

DIANNE：实际上，有两处不同。当ModelState.IsValid属性为真时，我们应该使用IRoomViewModelWriter接口来保存消息。但我们应该以JsonResult类返回视图模型。

DAVID：那么如果模型状态无效时，我们要做什么呢？

DIANNE：无效时我们应该返回一个带有HTTP 440错误的HttpStatusCodeResult类值。

DAVID：这是一个客户端错误响应代码吗？

DIANNE：是的，404是指错误请求的响应。

David给Dianne展示了他编写的代码，如代码清单11-14所示。

代码清单11-14　RoomController类上的AddMessage方法

```
[HttpPost]
public ActionResult AddMessage(MessageViewModel messageViewModel)
{
    ActionResult result;

    if(ModelState.IsValid)
    {
        writer.AddMessage(messageViewModel);

        result = Json(messageViewModel);
    }
    else
    {
        result = new HttpStatusCodeResult(400);
    }

    return result;
}
```

DAVID：IRoomViewModelWriter接口实现类中的AddMessage方法会是什么样子呢？

DIANNE：我想我们可以使用与IRoomViewModelWriter接口实现一样的模式。这部分代码并不在乎是否被异步调用，所以不需要改变我们目前使用的模式。

David按照IRoomViewModelWriter接口的CreateRoom方法中的模式编写了AddMessage方法，如代码清单11-15所示。

代码清单11-15　RoomViewModelWriter类上的AddMessage方法

```
public void AddMessage(MessageViewModel messageViewModel)
{
    var messageRecord = mapper.MapMessageViewModelToMessageRecord(messageViewModel);
    messageRepository.AddMessageToRoom(messageRecord.RoomID, messageRecord.AuthorName,
  messageRecord.Text);
}
```

至此，加上通过ADO.NET从Microsoft SQL Server获取消息记录的存储库方法后，最后一个故事也完成了开发工作。

现在David和Dianne已经完成了当前冲刺的所有故事，也刚好赶上了冲刺演示会议。

11.6 演示会议

星期五，团队安排了冲刺演示会议来为客户展示目前的开发进度。完成的故事被集成在了一起，并且要逐个展示每个故事的功能。团队希望客户可以基于此次演示提供反馈，以确定是否改变产品的开发方向。

冲刺演示会议有很多好处。第一，它能让团队确定开发工作始终是满足客户当前需求的。演示也能激励团队成员总是提供最好的产出，因为他们必须足够自信地向客户展示当前产品的开发状态。客户也能从冲刺演示会议中获取良多，因为他们在产品早期就能定期地看到产品上实实在在的进展。如果客户想要改变产品的工作方式，演示会议就是提出建议和意见的最佳场合。演示会议的输出是改变和重新排序后的用户故事积压工作，而且软件产品可以从下一个冲刺起立即根据客户需求改变开发方向。

Proseware 软件的第一个演示

在演示会议这天，整个团队和一个客户代表都齐聚在会议室。团队简单地讨论了当前冲刺的进展情况，并准备为客户展示当前产品的开发状态。

关键时刻，不幸的事情发生了，会议室的投影仪工作不太正常。这让演示推迟了几分钟，直到一个技术人员解决了投影仪的问题。投影仪重新启动后，团队发现屏幕分辨率比他们的开发时低好多。这样用户界面的表现差了很多，客户都很难分辨布局到底是在哪个页面上。

团队向用户道歉，客户也似乎表示理解，但是他还是抱怨了展示的应用程序并没有达到他们想要的标准。在反复调整了一些显示设置后，应用程序看起来好了点，但是依然达不到开发环境中展示的水平。

团队逐个展示了这个冲刺已经实现的四个故事，然后询问客户是否有问题或建议。客户的评价整体上是可以接受的：尽管应用程序还没做多少事情，但是一些核心功能在很早期就已经就绪可用了。

Petra建议接下来的任务应该包括格式化消息的发送。她也提到了David有关内容过滤器的想法。看起来客户对这个主意很感兴趣，并要求团队把它作为下一个冲刺的高优先级工作项。

会议结束后，客户先行离开了，整个团队依然留在会议室，准备开始他们的冲刺回顾会议。

11.7 回顾会议

冲刺结束时，所有团队成员都聚在一起讨论本周的工作进展，他们都要讨论和回答以下问题。

- 什么做得比较好？
- 什么做得不太好？

- ❑ 流程中有没有需要改进的部分？
- ❑ 冲刺中有没有需要保持的新东西？
- ❑ 冲刺中有没有发现任何意料之外的事情？

冲刺会议的目标是生成一个可执行的、排好序的工作项列表，然后在会后实施。按照惯例，会议的输出不是为了生成无法切实执行的结论。

11.7.1　什么做得比较好

会议室中，所有团队成员逐个回答上面问题列表中的每个问题。他们先从当前冲刺中做得比较好的事情开始。

STEVE：那么，大家认为第一个冲刺中做得比较好的都有什么？

PETRA：我认为演示不错。尽管没有完全达到客户的期望，但是效果依然是积极的。

TOM：我完全同意。我认为客户也清楚地知道这还不是最终的产品，甚至不是第一个要发布的版本，但是演示在短短的时间里让客户知道了已完成工作的实实在在的价值。

DIANNE：他们很积极地提供了有建设性的反馈，也改变了我们的开发方向，即使在一个非常早的阶段。

PETRA：是的，演示后我们还提到了另一个有价值的想法。David，你的内容过滤器的主意得到了用户很好的响应，他们提高了它的优先级，因为我们需要在下一个冲刺中开工实现它。

DAVID：我很高兴客户对我们正在做的事情表现得很热情。我们只需要管理一下他们的期望值：我不想他们过分表扬我们现有的工作成果。

STEVE：还有其他什么做得比较好的吗？

DIANNE：我认为代码写得不错。尽管现在代码量不大，但它们肯定为后续工作提供了一个很好的基础。

DAVID：我需要给Dianne和Steve说声谢谢，在和他们协作开发的过程中，我学到了很多。如果没有他们的指导，现在的代码中很可能已经包含了一些明显的技术债务。

STEVE：好的，我会记录这些优点。还有其他的吗？

会议室开始变得安静了，所以Steve在白板上记下了以下几点，并开始准备讨论下一个问题。

- ❑ 演示不错，即使在这个阶段没多少要展示的东西。
- ❑ 客户参加了演示会议并且提供了很好的反馈。
- ❑ 代码目前看还不错。

11.7.2　什么做得不太好

团队要回答的下一个问题是，这个冲刺中哪些做得不太好。

STEVE：这个问题我们要实事求是地回答：这个冲刺中什么做得不太好？

TOM：我觉得这个冲刺我没有忙起来。我发现这个阶段我并没有很多可以做出贡

献的工作。只有很少的东西可以测试，而且反复来了好多次。

PETRA：好的，这个反馈很有价值。那么原因是什么呢？

STEVE：我认为这是因为David是为唯一实现故事的开发人员，当时Dianne和我正在为另外一个项目准备一些架构的设计。

PETRA：那这个问题还一直有吗，或者说我们能解决它吗？

DIANNE：这个要记录在“有待改进”的事项列表中，但是Steve和我打算接下来在Proseware项目中要更像一个猪的角色，而不是鸡的角色。

TOM：能告诉我你的确切意思吗？

STEVE：大体上讲，我们虽然对Proseware项目有贡献，但是并没有完全投入。前面一段时间我们被要求做另外一个项目的维护工作。

DIANNE：的确是这样的。我们会在接下来的冲刺中和David一起实现用户故事，这样就能给你交付持续可测的功能，而不是比较凌乱的交付。

TOM：这样挺好。

STEVE：还有其他什么做得不好的吗？

PETRA：显然，演示会议开始时的投影仪问题可不太好。

DAVID：是啊，我真不知道投影仪哪里出问题了！当时觉得很尴尬，不过我想自己恢复得还算比较好了。

DIANNE：是啊，我们补救得还不错。但我们需要防止再犯这样的错误。

STEVE：我认为这个问题的根本原因就是缺乏足够的准备时间。我们需要更早更频繁地在所有可能的环境中集成测试！以后，我们需要在演示前花半个小时在会议室用投影仪预演我们展示的东西。客户不会在意一次这样的错误，但是总是犯同样的错误就不可以接受了。

Steve又在白板上添加了几个记录。

- ❑ 这个冲刺中测试没有足够的可测试内容。
- ❑ 演示差点因为环境问题搞砸。

STEVE：还有其他的吗？

所有团队成员都摇摇头表示没有，Steve决定开始讨论下一个问题。

11.7.3 什么需要改变

敏捷流程非常灵活，团队应该借着回顾会议的机会来说出流程哪些部分适合他们，哪些部分阻碍他们。为改进流程、工作环境或其他实践而制定可执行的工作计划是改进团队工作的一个很好的办法。

STEVE：现在，我们已经得出一些我们没做好的事情上的改进事项。第一，从下一个冲刺开始，Dianne和我自己将投入更多的精力到项目中来。第二，我们需要再拿出半个小时在会议室用投影仪进行预演。还有其他吗？

会议室很安静。

STEVE：那好吧，我来问。每日站立会议有什么问题吗？

TOM：实际上，是有问题的。这个冲刺因为堵车导致我缺席了两次站立会议，你还记得吗？

DIANNE：我也一样，我都忘记说这个了。

STEVE：那改为早上九点半会好些吗？

TOM：嗯，会好些。对我来说，真的无法保证早上九点就能到办公室。

PETRA：我会通知管理层，我们应该实行弹性工作制度。

TOM：谢谢，那样最好。

DAVID：我认为我们应该给客户的演示间隔时间更长点。太频繁的话，需要客户每周来到我们这里，但每次就只能看到不太多的进展。两周一次可以解决这两个问题，你们觉得呢？

STEVE：这个主意很好，肯定可以解决你说的问题。可是我认为演示是激励我们交付的一个很好的动力。我依然想保持每周演示的频率，那么对客户的演示每两周一次，我们自己内部的演示依然是每周一次，如何？内部演示只需要我们团队自己参加，不需要管理层参加。

PETRA：我觉得这样是可行的。这样能让我们完善我们的演示流程，同时也能让客户一次看到更多的功能呈现。

David很高兴他的意见得到了采纳，Steve在白板上再加上了下面几点。

- ❑ Dianne和Steve需要为交付用户故事投入更多的精力到项目中。
- ❑ 给客户的演示每两周一次。
- ❑ 内部演示只需要团队参加。
- ❑ 正式演示前，安排额外的半个小时来回顾演示计划并进行预演，以确定一切都已准备就绪。

11.7.4　什么需要保持

有时，最好的行动就是不行动，换句话说，就是要把团队已经做了但还没有形成习惯的事情记录下来。这是什么需要保持这个问题的主题。

STEVE：那么，有什么需要保持的吗？有什么我们在这个冲刺首次采用且应该继续保持下去的吗？

每个人依然不说话。

STEVE：好吧，如果有人会议之后想起任何需要保持的事情，请告诉我，我们也会为它们制定行动计划。

11.7.5　遇到了什么意料之外的事情

几乎每个冲刺我们都会遇到一些意料之外的事情。团队可能会发现一个旧的流程需要更新，

有需求被遗漏，软件的一部分功能突然又不工作了等。回顾会议中找出这些意外的目的是让团队以后碰到这些事情时不再感到意外。

STEVE：大家有没有遇到什么意料之外的事情？

DAVID：我很意外客户对我提出的内容过滤器很感兴趣。

DIANNE：说实在的，你的主意很有意义。客户对Proseware的定位是一些特定的人群，内容过滤很可能会给他们带来很多价值。

PETRA：我同意Dianne说的，这的确是一个很棒的主意。所以我们真的应该继续保持一些事情，比如继续保持对新想法的思考。

Steve在白板上需要保持事项一列中添加了一条记录。

TOM：我很惊讶，在这个项目上Steve和Dianne竟然没有专职工作。

PETRA：是啊，其他人是否也觉得意外？

DIANNE：这也让我感到意外。我给别人的印象是我是专职完成Proseware项目的，但是实际上Steve和我却被安排去帮另外一个项目救火。

STEVE：我们应该和管理达成某些协议，以保证下一个冲刺中去征调其他人，而不是我们。

DIANNE：这个计划好。

Steve在白板上记下了这个行动计划。

STEVE：好了，谢谢大家。这个冲刺整体很不错，给项目开了一个好头。让我们继续保持。

PETRA：我赞同Steve的总结。让我们开始执行这次回顾会议所达成的行动计划，并且确保我们下一个冲刺也做这样的回顾和计划。

会议圆满结束，团队成员都离开了会议室。

11.8 总结

对于Proseware项目的团队而言，第一个冲刺是成功的。尽管不是所有事情都按照计划进行，但团队已经早早收集到了很多有价值的反馈。这是任何敏捷流程的关键：总是要为建设性的批判采取纠正措施。

下一章中，团队会继续进行第二个冲刺以实现剩余的用户故事。

第 12 章

自适应实例冲刺2

完成本章学习之后，你将学到以下技能。

- 观摩团队第二个冲刺中计划会议的过程。
- 追踪剩余用户故事的实现和演进过程。
- 观摩团队第二个冲刺中演示和回顾会议的过程。

本章中，Trey Research团队会继续为Proseware项目实现剩余的用户故事。根据第一个冲刺中回顾会议上用户的反馈，团队对开发方向做了些微的调整，并为第二个冲刺设置了下面的冲刺目标。

> 会话中可以发送格式化文本，可以过滤消息内容以确保它是合适的，确保可以同时支持三百个用户。

这个冲刺承诺完成剩余的所有故事。按照惯例，团队会在冲刺收尾时的演示会议上给用户做演示，以向用户展示产品开发的进度并收集用户的反馈。在接下来的回顾会议上，团队还会讨论冲刺期间做得好的事情和做得不太好的事情。

团队首先从这个冲刺的计划会议开始。

12.1 计划会议

在项目第二个冲刺的计划会议上，团队成员会一起讨论与这个冲刺目标相关的用户故事。第二个冲刺包括以下用户故事。

- 我想发送正确格式化的标记。
- 我想过滤消息内容以确保它是适合发表的。
- 我想同时服务数百个用户。

所有人都在会议室就坐后，会议就开始了。

> PETRA：根据上个冲刺的演示会议上客户的反馈，我们的用户故事积压工作上多了一项。
>
> STEVE：客户想要我们先实现内容过滤的故事，以取代原计划的只读会话的故事。
>
> DIANNE：所以我们应该先估算这个新的故事。
>
> STEVE：是的，我们完成估算后才能知道在这个冲刺中我们的开发资源还有多少。

DIANNE：好的，那我们就在数到三的时候，一起亮出自己的点数估算扑克。

每个人在亮出扑克之前都掩盖着它。

PETRA	TOM	STEVE	DIANNE	DAVID
3	3	8	5	8

STEVE：哇哦，我们的估算差异比较大。Tom，你能解释一下为何是三个点吗？

TOM：我选择三个点，主要因为我可以很容易自动化测试这个故事。这个根据受限词汇库来判断文字信息是否可以发送的用例是比较简单的。如果用更复杂的技术实现，我也愿意增加估算值。

STEVE：David，你能解释一下你的八个点的估算吗？我也会随后给出我的理由。

DAVID：好的。我认为实现会比较困难，因为我们需要为受限词汇增加一个新的数据表，而这是需要一定的时间才可以实现的。

STEVE：是的，我也是这样想的。我还考虑到，我们不能只想限制用户会话中的消息文字，也应该检查他们给房间起的名称。实际上，用户的任何输入都应该经过内容过滤检查后才能提交并发表。

DIANNE：现在我们可以先实现一个由数据驱动的内容过滤器，把方案简化为使用固定的词汇列表，怎么样？

STEVE：好的，这个主意不错。后面我们再专门针对内容过滤器的管理增加相应的用户故事。

PETRA：好，那我们是要重新估算，还是直接按照五个点的估算来？

TOM：我觉得五个点可以。

STEVE：嗯，五个点挺合适。

DAVID：我同意，就按照五个点来。

PETRA：好的，谢谢大家。让我们一起努力确保完成这个冲刺的所有目标吧，这样就可以在这周给客户做一个精彩的演示。

大家陆续走出了会议室，准备开始动工了。

12.2 “我想发送正确格式化的标记”

在开始实现故事之前，David向Steve请教有关解析标记的事情。

DAVID：我觉得我们应该使用一个第三方库来完成解析标记并转换为HTML的工作。但是我不确定要使用哪个库。

STEVE：好的，等一等。我知道Dianne有这方面的经验。让我们一起请教她吧。

Steve招呼Dianne过来，并请教她有关标记库的事情。

STEVE：Dianne，你以前使用过一些标记库，对吧？你觉得哪个好呢？

DIANNE：我以前评估过一些这方面的第三方库。试一试MarkDownDeep，它应该

可以满足我们的需求了。大家可以通过NuGet获取它的包。

STEVE：谢谢你，Dianne。David，那我们就试一试MarkDownDeep吧。还有，要确保创建一个我们自己的库对MarkDownDeep进行适配，这样做，项目的其他库就都只需要依赖这个我们自己的适配库。

DAVID：好的，谢啦。

David开始着手为项目实现标记传输的功能。过了几个小时后，他做好了给Steve和Dianne展示自己实现的准备，然后他像第一个冲刺一样请求他们一起给他的代码做同行评审。

DAVID：我想知道标记传输实现的最佳位置是哪里。我知道我可以通过为现有接口增加一个修饰器实现来截获房间的消息文本。我想，如果在`IMessageRepository`接口的`AddMessageToRoom`方法中实现它的话，我们就可以不用在读取时处理标记了。如果我们只是在保存消息时将标记转换为HTML，我们就不需要再操心标记的事情了。

DIANNE：这样的确可以避免每次读取时进行标记到HTML的转换，但实际上它不能正常工作。

DAVID：是的，我也认识到如果我们在保存消息时进行转换的话，就不能再编辑消息了。我知道我们现在还没有这个特性，但是我想将来可能会有这个需求，我不想让用户无法编辑消息。

STEVE：很好。其实我们几乎可以肯定后期会有你所说的需求，所以应该在读取时处理标记转换，而不是在编写消息时完成这个动作。

DAVID：我的另外一个问题是有关转换的位置，是在客户端进行转换，还是在服务器端进行转换？我现在暂时选择在服务器端实现。但是我想知道，是否应该在浏览器的客户端完成这个动作？

DIANNE：或许我们能基于这个本地转换功能为用户提供这样一个特性：用户在输入消息时能即时预览标记和HTML格式的显示。

STEVE：好主意，Dianne。我会记下这个，然后在演示会议上征求客户的意见和想法。

DAVID：最后，我是将转换功能作为`IRoomViewModelReader`接口的一个修饰器实现。因为标记和HTML都只是用户界面相关的概念，`IRoomViewModelReader`接口又是一个用户界面的契约。还可以用转换修饰`IRoomRepository`接口或`IMessageRepository`接口，但这两个都是数据契约。还有，我对自己写的代码不是完全满意，你们帮我看看。

David给Steve和Dianne展示了他实现的标记修饰器，如代码清单12-1所示。

代码清单12-1　标记修饰器可以将用户输入的标记转换为HTML

```
public class RoomViewModelReaderMarkdownDecorator : IRoomViewModelReader
{
    public RoomViewModelReaderMarkdownDecorator(
        IRoomViewModelReader @delegate,
        Markdown markdown)
```

```
    {
        this.@delegate = @delegate;
        this.markdown = markdown;
    }

    public IEnumerable<RoomViewModel> GetAllRooms()
    {
        return @delegate.GetAllRooms();
    }

    public IEnumerable<MessageViewModel> GetRoomMessages(int roomID)
    {
        var roomMessages = @delegate.GetRoomMessages(roomID);

        foreach(var viewModel in roomMessages)
        {
            viewModel.Text = markdown.Transform(viewModel.Text);
        }

        return roomMessages;
    }

    private readonly IRoomViewModelReader @delegate;
    private readonly Markdown markdown;
}
```

STEVE：对我来说，你的实现挺好的。你觉得什么地方不满意呢？

DAVID：有两个地方我不满意。第一，我们直接依赖了MarkdownDeep库的Markdown类。这个不是应该隐藏到一个单独的接口后吗？

DIANNE：我觉得给修饰器直接注入该类的依赖是可以接受的。因为Markdown类相当小，我们后期要更换成另外一个库的工作量也不大。

DAVID：不大就好。直接使用Markdown类也能让我的针对期待转换结果的单元测试更简单些。这是我写的单元测试代码。

David打开了包含他编写的标记单元测试代码的文件，如代码清单12-2所示。

代码清单12-2　针对标记转换修饰器的单元测试

```
[TestFixture]
public class MarkdownTests
{
    [Test]
    [TestCase(
        "This message has only paragraph markdown...",
        "<p>This message has only paragraph markdown...</p>\n")]
    [TestCase(
        "This message has *some emphasized* markdown...",
        "<p>This message has <em>some emphasized</em> markdown...</p>\n")]
    [TestCase(
        "This message has **some strongly emphasized** markdown...",
        "<p>This message has <strong>some strongly emphasized</strong>
  markdown...</p>\n")]
    public void MessageTextIsAsExpectedAfterMarkdownTransform(string markdownText,
```

```
  string expectedText)
    {
        message1.Text = markdownText;
        var markdownDecorator = new
  RoomViewModelReaderMarkdownDecorator(mockRoomViewModelReader.Object, markdown);

        var roomMessages = markdownDecorator.GetRoomMessages(12345);

        var actualMessage = roomMessages.FirstOrDefault();

        Assert.That(actualMessage, Is.Not.Null);

        Assert.That(actualMessage.Text, Is.EqualTo(expectedText));
    }

    [SetUp]
    public void SetUp()
    {
        markdown = new Markdown();
        message1 = new MessageViewModel
        {
            AuthorName = "Dianne",
            ID = 1,
            RoomID = 12345,
            Text = "Test!"
        };
        mockRoomViewModelReader = new Mock<IRoomViewModelReader>();
        var roomMessages = new MessageViewModel[]
        {
            message1
        };
        mockRoomViewModelReader.Setup(reader =>
                reader.GetRoomMessages(It.IsAny<int>())).Returns(roomMessages);
    }

    private MessageViewModel message1;
    private Mock<IRoomViewModelReader> mockRoomViewModelReader;
    private Markdown markdown;
}
```

STEVE：单元测试的实现也挺好啊。你不是说还有一个地方有所顾虑吗？

DAVID：是啊，你们注意一下修饰`IRoomViewModelReader`接口的标记类型，它也有一个`GetAllRooms`方法。这里，是不是可以做接口分离？因为它的`GetAllRooms`方法只是直接委托被包装的实例来完成实际的工作而已。

DIANNE：我们也应该允许用户在房间名称里使用标记吧？如果允许的话，那`GetAllRooms`方法也会需要做修饰的动作了。

STEVE：我认为代码实现可以保持现状。我们先暂时不要做接口分离，也暂时不允许房间名称使用标记。后面我们可以基于演示会议上客户的反馈再做决定。

12.3 “我想过滤消息内容以确保它是适合发表的”

Dianne和David都被安排来实现消息内容过滤的用户故事。他们再次通过结对编程来一起实现这个故事所需的功能。

DIANNE：按照我们在计划会议上的讨论结果，这个冲刺，我们先不去实现一个完整的由数据驱动的消息过滤器。但是，我们需要为将来切换到完整的数据过滤器打好基础。

DAVID：那么这个冲刺就只实现数据访问部分，然后下周再继续剩余部分？

DIANNE：不，我们不能这样做。我们依然需要交付一个竖切可用的功能，它是可以给用户演示的东西，但可以不完整。如果我们只实现数据访问部分，这不会给客户带来任何价值。

DAVID：我不太明白你说的。既能给用户展示价值，但又不是完整的内容过滤器？

DIANNE：我们要做的是在某个地方采用折中的方案以便可以短时间实现，但依然在整体上可以给客户提供一些价值。对于这个故事，需要折中的地方很清楚：受限词汇的列表应该直接硬编码，而不是从诸如数据库等持久存储中读取。

DAVID：我觉得这个是可以的。但是，据我了解，任何硬编码都是不太好的。应用硬编码的都是不好的方案，对吗？

DIANNE：一定程度上，是的。这是一个技术债务。我们是在慎重地做出使用折中方案的决定，以便能够尽早交付一些东西。也许客户知道自己想要的受限词汇清单而且永远不会改变。如果真是这样，我们就可以用这样的硬编码清单来完成故事。

DAVID：我觉得这招很好，真的。我们把更简单的方案看作是实现最终目标路上的一个中间目标，而不是直接耗费大量时间来实现一个东西。

DIANNE：正是这样！而且，每当达成一个中间目标时，接下来的中间目标也许会变得截然不同，甚至最终目标也可能会发生彻底的改变。

DAVID：另一个我不太确认的事情是：我们应该如何实现这个故事？为消息编写器创建一个修饰器，当消息中有受限词汇时就引发异常？

DIANNE：你这个建议的问题在于把异常用在正常的控制流程中了。异常最好只用于真正意外的情况。现在这个故事更多的是一个验证场景。

DAVID：哦，我明白了。所以我们应该勾入MVC框架中的验证机制，当消息包含了受限词汇时就让验证失败？

DIANNE：嗯，你这个主意不错。那么我们这样实现……？

Dianne直接开始编写代码，过了十几分钟，创建了如代码清单12-3所示的类。

代码清单12-3　自定义验证属性非常适合内容过滤器

```
public class ContentFilteredAttribute : ValidationAttribute
{
    private readonly string[] blockedWords = new string[]
    {
```

```
        "heffalump",
        "woozle",
        "jabberwocky",
        "frabjous",
        "bandersnatch"
    };

    protected override ValidationResult IsValid(object value,
  ValidationContext validationContext)
    {
        var validationResult = ValidationResult.Success;

        if (value != null && value is string)
        {
            var valueString = (string)value;
            if(blockedWords.Any(inappropriateWord =>
                    valueString.ToLowerInvariant()
                    .Contains(inappropriateWord.ToLowerInvariant())))
            {
                var errorMessage = FormatErrorMessage(validationContext.DisplayName);
                validationResult = new ValidationResult(errorMessage);
            }
        }

        return validationResult;
    }
}
```

DIANNE：我专门使用了一些明显不真实的受限词汇，因为我不想给用户的演示包括任何真正的受限词汇。

DAVID：嗯，当然。我认为只要很好地展示了功能即可。现在我们开始写单元测试？

DIANNE：好，单元测试是这样的。

代码清单12-4展示了两个针对RoomControllerTests类的单元测试。

代码清单12-4　添加单元测试是为了强制执行房间名称和消息文本上的验证规则

```
[Test]
[TestCase("Callooh! Callay! O frabjous day!")]
[TestCase("The frumious Bandersnatch!")]
[TestCase("A heffalump or woozle is very confusel...")]
public void PostCreateNewRoomWithBlockedWordsCausesValidationError(string roomName)
{
    var controller = CreateController();

    var viewModel = new RoomViewModel { Name = roomName };
    var context = new ValidationContext(viewModel, serviceProvider: null, items: null);
    var results = new List<ValidationResult>();

    var isValid = Validator.TryValidateObject(viewModel, context, results, true);

    Assert.That(isValid, Is.False);
```

```
}
// . . .
[Test]
[TestCase("Callooh! Callay! O frabjous day!")]
[TestCase("The frumious Bandersnatch!")]
[TestCase("A heffalump or woozle is very confusel...")]
public void PostAddMessageWithBlockedWordsCausesValidationError(string text)
{
    var controller = CreateController();

    var viewModel = new MessageViewModel { AuthorName = "David", Text = text};
    var context = new ValidationContext(viewModel, serviceProvider: null, items: null);
    var results = new List<ValidationResult>();

    var isValid = Validator.TryValidateObject(viewModel, context, results, true);

    Assert.That(isValid, Is.False);
}
```

DAVID：使用视图模型中带有ContentFiltered标记的两个属性，这些测试会很快成功通过验收的。

DIANNE：当然。只是我依然觉得现在的设计有些部分不顺眼。

DAVID：哪些部分？

DIANNE：很明显的是，硬编码的受限词汇列表。虽然我们已经决定把它看作技术债务，但是我依然想要从一个提供者接口中获取这些受限的词汇清单。按照这个思路，我可以创建一个临时实现类用于返回一个静态的硬编码的受限词汇列表，这就为以后真正的数据驱动实现提供了扩展点。

DAVID：是的，这样就可以把受限词汇列表数据从验证算法中分离出去。我们可以在这里使用依赖注入吗？

DIANNE：很不幸，不可以。这些自定义属性并不是很灵活，因为不可以从控制器或者MVC提供的其他扩展点创建它。

DAVID：很可惜啊。那么服务定位器模式怎么样？

DIANNE：我通常会把它看作反模式，但是现在这个用例中，服务定位器应该是可用的最佳选择了。

DAVID：我们是否应该保持现有的代码结构，就把这些属性标记为技术债务，可以吗？这样做，我们就可以继续开发，也可以在需要的时候返回重构这部分代码。

DIANNE：我同意。我认为这里有关受限词汇的需求很可能会在将来发生彻底的变化，而且目前我们无法猜测它会朝哪个方向发展。

12.4 “我想同时服务数百个用户”

冲刺的最后一个故事要求Dianne和Steve增强应用程序的扩展性。客户需要应用程序支持水平扩

展，而不是重置扩展。水平扩展是指应用程序应该能够通过额外的服务机器以支持更多用户的并发访问。与此相比，垂直扩展是指应用程序在通过增强单个机器的能力以支持更多用户的并发访问。

Dianne和Steve明白水平扩展的受限点在于诸如Microsoft SQL Server等关系型数据库管理系统（Relational Database Management System，RDMS）的架构。与本身就支持水平扩展的带有专门设置的分布式存储方案相比，现有的架构很难在服务集群中的另外一台机器中增加一个新的SQL Server实例。

基于这样的认识，Dianne研究了可供替代SQL Server数据库的各种文档存储的候选项。她向团队推荐了MongoDB，认为它是一个可靠且流行的，能很好地通过增加新服务器来扩展应用程序能力的文档存储系统。现在唯一的问题是，应用程序是从SQL Server存取房间和消息数据的。

幸好团队已经通过面向接口编程为存储架构的改变提供了扩展点。

> DIANNE：我们要做的就是为房间和消息存储库这两个接口创建一个新的实现。
>
> STEVE：嗯，我肯定可以那样做，但是我认为可以把记录数据到视图模型的数据转换直接干掉，取而代之的是，直接序列化和反序列化我们的视图模型。
>
> DIANNE：听起来有点意思！我们要做的只是为`IRoomViewModelReader`和`IRoomViewModeWriter`两个接口创建直接使用MongoDB的新实现。
>
> STEVE：说得很对。

两个人开始实现`MongoRoomViewModelStorage`类，如代码清单12-5所示。

代码清单12-5　MongoDB的数据持久性层的实现

```
public class MongoRoomViewModelStorage : IRoomViewModelReader, IRoomViewModelWriter
{
    public MongoRoomViewModelStorage(IApplicationSettings applicationSettings)
    {
        this.applicationSettings = applicationSettings;
    }

    public IEnumerable<RoomViewModel> GetAllRooms()
    {
        var roomsCollection = GetRoomsCollection();
        return roomsCollection.FindAll();
    }

    public void CreateRoom(RoomViewModel roomViewModel)
    {
        var roomsCollection = GetRoomsCollection();
        roomsCollection.Save(roomViewModel);
    }

    public IEnumerable<MessageViewModel> GetRoomMessages(int roomID)
    {
        var messageQuery = Query<MessageViewModel>
                .EQ(viewModel => viewModel.RoomID, roomID);
        var messagesCollection = GetMessagesCollection();
        return messagesCollection.Find(messageQuery);
```

```
        }

        public void AddMessage(MessageViewModel messageViewModel)
        {
            var messagesCollection = GetMessagesCollection();
            messagesCollection.Save(messageViewModel);
        }

        private MongoCollection<MessageViewModel> GetMessagesCollection()
        {
            var database = GetDatabase();
            var messagesCollection = database.
                    GetCollection<MessageViewModel>(MessagesCollection);
            return messagesCollection;

        }

        private MongoCollection<RoomViewModel> GetRoomsCollection()
        {
            var database = GetDatabase();
            var roomsCollection = database.GetCollection<RoomViewModel>(RoomsCollection);
            return roomsCollection;
        }

        private MongoDatabase GetDatabase()
        {
            var connectionString = applicationSettings.GetValue(MongoConnectionString);
            var client = new MongoClient(connectionString);
            var server = client.GetServer();
            return server.GetDatabase(ProsewareDatabase);
        }

        private readonly IApplicationSettings applicationSettings;
        private static string MongoConnectionString = "MongoConnectionString";
        private static string ProsewareDatabase = "Proseware";
        private static string MessagesCollection = "messages";
        private static string RoomsCollection = "rooms";
    }
```

完成这个故事之后，团队就不会再因为为Proseware应用程序和客户提供水平扩展能力的局限而受到束缚了。

12.5 演示会议

第二个冲刺的收尾阶段，团队为客户准备了另外一次演示。会议上会整理并展示这个冲刺完成的每个故事的功能。从第一个冲刺回顾会议总结的一个关键行动就是改善演示会议的准备情况。团队认真地做了准备，在没有客户参与的情况下，在会前完整地进行了一次功能展示的预演。这个过程有助于减少因为开发和演示环境不同所造成的问题。预演过程很顺利，所以团队做好了给客户展示项目进度的准备。

展示标记故事时，团队向客户代表征求了意见，问他是否也想要房间名称和消息内容可以使用同样的格式化标记。客户似乎接受了这个建议，并且让团队把这个也加入到产品积压工作中，但同时也表明这只是一个较低优先级的故事，因为他很快会有更重要的特性需求。客户也很高兴团队使用了简单的标记语言，而不是原来要求的HTML格式，因为客户也了解到标记语言是一种日益流行、友好随意的文本格式化方式。

下一个展示的故事是消息内容过滤。对于团队已经给房间名称和消息文本都应用了内容过滤器，客户也赞同以后所有接受用户输入的地方都要应用同样的文本过滤器。然而，客户提出了一个额外的需求，就是以后能通过配置来启用和禁用文本过滤器的功能。

在测试冲刺的最后一个故事时，Tom一次模拟了300个用户来操作分布在两台独立机器上的数据。客户再一次要求能用配置来控制数据源。

看完所有故事的展示后，客户表达了他对团队的赞赏，他非常高兴能看到团队能在短短的两个单周的冲刺里逐步取得了这么多切实可见的进展。

12.6　回顾会议

与第一个冲刺结束时一样，第二个冲刺结束时，团队也聚集在一起讨论过去一周的进展。所有团队成员都要讨论和回答以下问题。

- 什么做得比较好？
- 什么做得不太好？
- 流程中有没有需要改进的部分？
- 冲刺中有没有需要保持的新东西？
- 冲刺中有没有发现任何意料之外的事情？

冲刺会议的目标是生成一个可执行的、排好序的工作项列表，然后在会后实施。按照惯例，会议的输出不是为了生成无法切实执行的结论。

在一个舒适的会议室里，团队成员对列表中的问题进行逐一讨论。

12.6.1　什么做得比较好

团队首先讨论了这个冲刺中做得比较好的事情。

> STEVE：大家认为我们这个冲刺中什么做得比较好？
>
> PETRA：个人意见，我认为这个冲刺很成功：我们按时完成了冲刺目标，演示会议的准备工作也做得很充分，通过准备保证了演示的整个过程都很成功。客户也很满意，我们也对自己的付出所获得的成果感到满意。
>
> DIANNE：我同意。这个冲刺可以作为后续冲刺执行的一个模板。但是我们也不能忘记重点：我们必须要朝着能够长期保持这种水平而努力。
>
> STEVE：那当然，大家一定不要骄傲自满。

12.6.2 什么做得不太好

团队接着讨论这个冲刺中做得不好的事情。

STEVE：那么什么没做好呢？肯定有事情并没有按照计划进行。

DAVID：我认为有些问题在冲刺中一直处于没有答案的状态。这些问题对实现整体的影响不大，但是我认为应该尽快确定这些问题的答案，而不是一直等到冲刺结束。

STEVE：什么样的问题呢，David？你能举个例子吗？

DAVID：在标记转换的故事里，Dianne曾经问我们，房间名称是否也要像消息一样做转换。我选择不去做，但是我们实际上并不知道明确的答案。

12.6.3 什么需要改变

团队成员接着讨论需要改善的流程和工作实践的有关事项。

DIANNE：是的，我同意。我认为我们需要做些改进：如果我们有关于实现的问题，就应该去跟Petra确认。如果她也不肯定，她会安排与客户进行电话会议，并直接和客户沟通。

PETRA：这是当然，我在这里就是为你们获取任何你们所需的客户需求细节。如果你没有问过我，我也没有在故事中指定足够清晰的验收标准，那一定是有关键的东西被遗漏了。

STEVE：David说的这个问题，一直等到演示会议上再去询问用户并不会对我们有多大影响，但是对于其他一些更重要的问题，我们应该尽早找到正确的答案。

PETRA：还有什么需要改进的吗？如果没有，我们就开始讨论下一个话题。

12.6.4 什么需要保持

团队紧接着开始总结那些需要培养成习惯的积极行动。

STEVE：我要说些我们需要保持的东西。我们改进了演示的准备流程，并且取得了很好的效果。我们应该把这个记下来并坚持，直到形成习惯。

PETRA：说得好。下一个冲刺会证明这个冲刺工作质量能否继续保持，或者只是个意外，所以记录我们如何完成这个优秀的冲刺并且在以后总是将这个样板过程牢记在心会是一个很好的主意。

STEVE：还有什么我们要保持的？

DIANNE：我想不到还有什么我们特别需要保持的东西了。

12.6.5 遇到了什么意料之外的事情

回顾会议中找出这些意外的目的是让团队以后碰到这些事情时不再感到意外。

STEVE：最后一个问题：这个冲刺中有没有我们后续需要研究或防止出现的意外事项？

DAVID：我对这个冲刺进行得这么好感到有些意外！

团队结束了这次回顾会议，所有成员都陆续走出了会议室，满脸洋溢着对冲刺成功的喜悦。

12.7　总结

第二个冲刺很成功，对团队而言是一个改进。通过在选择方案中引入自适应代码，团队展示了在敏捷软件开发项目中优雅地处理各种需求变更是可以做到的。如果团队没有在源代码中引入扩展点，团队成员要增强软件功能会非常困难，需要大量的重写、重构，或者将已有代码弄得支离破碎。

附 录

自适应工具

本附录为你简要介绍如何使用Git进行源代码控制。使用Git可以获得本书的示例代码。如果你以前使用过Git，那么一定很清楚这个源代码控制软件如此出名也是应该的。如果你以前没有使用过它，本附录会为你讲解如何使用Git管理本地和远程存储库。这些技能可以应用在任何存放在Git上的存储库；因此这里讲解的内容并不局限于本书的代码示例。有很多著名的开源项目都在使用Git，也有很多公司开始使用Git来管理他们自有的代码资产。

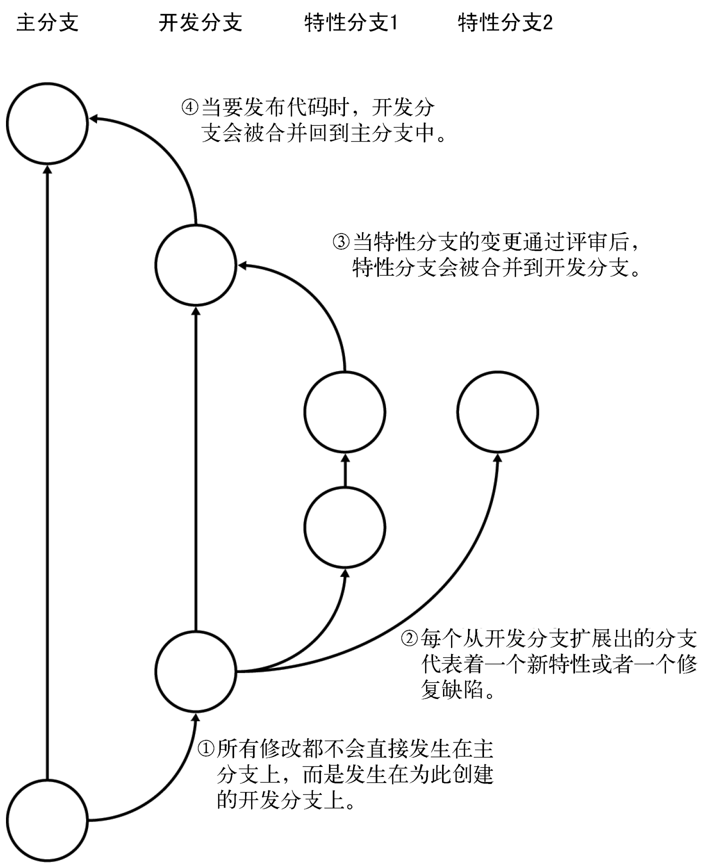
不论具体实现如何，持续集成（Continuous Integration，CI）系统都是一个能让不同参与者之间的代码始终保持同步的重要工具，所以本附录中有一节简要讨论了持续集成的概念，并介绍了一个实施持续集成的通用工作流程。

A.1 使用 Git 做源代码控制

在诸如Mercurial和Git这样的分布式源代码控制系统出现之前，源代码控制的理论和实践在很长时间里发展得都很慢。使用任何源代码控制系统总是比不使用好，但我肯定首选Git。

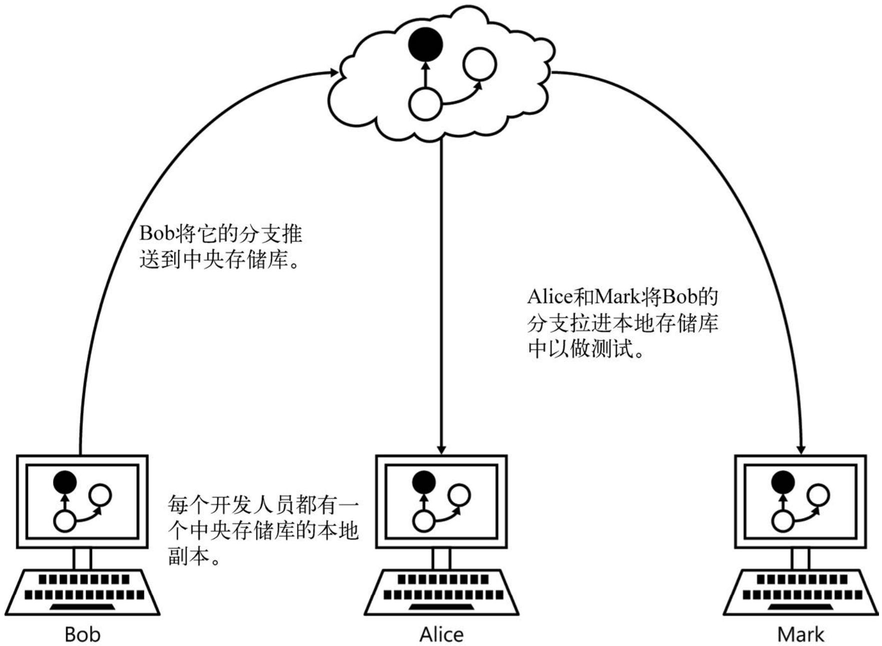
通常来说，源代码控制系统就是为了追踪代码随着时间推移时所发生的变更，通过它能够很容易地按照时间线前后遍历代码的变更，而且它同时也能提供一份只读的源代码备份。

使用Git，每个开发人员都有自有的、包含所有源代码的存储库（详见图A-1）。为了编辑源代码，开发人员应该为提交连续的变更创建本地分支。每个分支都有一个清楚描述的目的：修复某个缺陷，实现某个新特性，或改进某些体验。不论目的是什么，这些变更都只在本地的存储库中发生，直至开发人员决定把代码推送到其他的远程分支中。



图A-1　一种Git分支策略

尽管理论上专门设置一个中央存储库是没有必要的，但实际应用中通常都会选择某个存储库作为正式的源代码存放位置，参见图A-2。在一个开发人员将本地分支推送到这个中央存储库后，其余的开发人员都陆续接到请求以将该分支包含的新变更拉进他们本地存储库的主分支中。这被称作拉请求（pull request），并且经常用在有助于保证代码质量的源代码同行评审活动中。每个代码评审者可以根据具体情况接受或者拒绝评审中代码相关的拉请求，也可以将分支拉进自己的本地存储库中进行编译和测试。如果该开发者提交的代码变更被拒绝了，他可以继续修改并再次将新的变更推入中央代码库直至获得批准。然后，每个获得批准的拉请求都会被合并到一个主开发分支中，其他的开发人员在下次将主分支代码同步到本地存储库时会获取到所有已经获得批准的拉请求中包含的变更。如果需要，他们也要将这些变更合并到自己本地正在进行的分支中。



图A-2　分布式源码控制是一种点对点系统，但实际应用时通常会设置一个中央代码库

A.2　Git 课程

支持Windows平台的Git安装包可以从http://git-scm.com/download/win下载。

本书中所有的代码清单都是存放在GitHub上的。GitHub是一个Git存储库社区的中心存储库。

下面几节会为Git的初学者简要介绍如何浏览Git存储库上的代码。相比完整的Git介绍，这几节提供的内容还远远不够，但是你至少可以获取本书代码示例并在本地进行编译。要获得更多的Git使用帮助，Git参考手册[1]会是一个很好的完整介绍。

如果不太喜欢使用命令行，http://git-scm.com/downloads/guis上也有好几款不错的带有用户界面的Git工具，你可以选择下载自己喜欢的。在我写这本书时，用户下载最多的是Atlassian的SourceTree。

A.2.1　克隆存储库

获取Git存储库中代码的第一步就是克隆指定的存储库。所有Git动作的命令都是git命令行应用程序的参数。其中`clone`命令需要提供需要克隆的目标存储库的位置信息。下面的命令示例可

① http://gitref.org。

以将本书的存储库克隆到本地。记住，Git是一个分布式源代码控制系统，所以会有很多的存储库。对远程存储库你只有读取权限，但对克隆得到的本地存储库你是有改写权限的。

```
git clone https://github.com/garymcleahhall/AdaptiveCode.git
```

这个命令会在你本地的工作目录下生成一个新的名为AdaptiveCode的子目录。默认选择的是master分支。但是，本书的每个示例都安排在不同的分支上，因此你也需要知道如何切换分支。

A.2.2　切换分支

克隆好新的存储库后，可以使用改变目录命令来改变你本地存储库的目录。

```
cd AdaptiveCode
```

对于本书存储库，当前默认选择的分支是master。但master分支上又没有多少东西，代码都安排在其他的分支上。起初，只将master分支克隆到了本地，其余的分支都没有从远程存储库克隆到本地。通过给git提供branch命令，可以查看本地可用的分支清单。

```
git branch
```

这个命令只列出master分支。为了列出所有远程的分支，还需要在branch后提供remote命令。

```
git branch –remote
```

这个命令可以列出本书存储库中所有的分支。注意，所有分支都以origin/前缀开头，用来表明这些分支所在的远程位置。每个存储库可以同时有多个远程位置，使用诸如origin的名称可以将指定的远程存储库克隆到本地。

尽管分支几乎可以使用任意名称，但作为个人喜好，我还是为本书存储库的每个分支选用了chX-这个名称前缀。chX表示分支相关的章节编号，名称的其余部分是对分支内容的简短描述。附录B提供了一个包括代码清单及其他们相应的分支名称的参考清单。现在，通过checkout命令，你可以创建远程分支的本地副本并切换到它上面去。

```
git checkout ch9-problem-statement
```

执行这个命令会创建远程分支origin/ch9-problem-statement的本地版本，并且将当前工作目录切换至这个分支上，以便后续改动都只作用于相应的本地分支。列举当前工作目录的内容，你会看到如下清单所示的内容，其中有一个名为DependencyInjectionMvc的新目录，它包含了一个Microsoft Visual Studio解决方案文件以及一些该方案所包含子项目的子目录。

```
C:\dev\AdaptiveCode [ch9-problem-statement]> ls


    Directory: C:\dev\AdaptiveCode


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         3/16/2014  12:47 PM            DependencyInjectionMvc
-a---         3/16/2014  12:47 PM       1522 .gitignore
-a---         3/16/2014  12:30 PM         84 README.md
```

如果你切换回master分支，这个文件夹会被删除，因为它与当前选择的分支已经无关了。

A.2.3 更新本地分支

如果某个时候分支的远程版本有了变更，你也想获取这些最新的变更，那么使用fetch命令就可以下载远程分支的所有变更。

```
git fetch
```

如果你不指定分支名，这个命令会下载所有分支，包括那些新创建的分支。你也可以通过指定分支名来下载想要获取的分支。

```
git fetch origin master
```

注意，上面的命令中也指明了远程存储库的名称，因为master分支很可能在多个远程存储库中都存在。

使用fetch命令下载分支变更后，你可以使用checkout命令切换至目标分支。

```
git checkout ch9-problem-statement
```

从这里开始，本地分支就不再与远程分支同步，因为远程变更并没有再被克隆到本地。使用merge命令可以将远程分支上的所有变更合并到本地分支中。

```
git merge origin/ch9-problem-statement
```

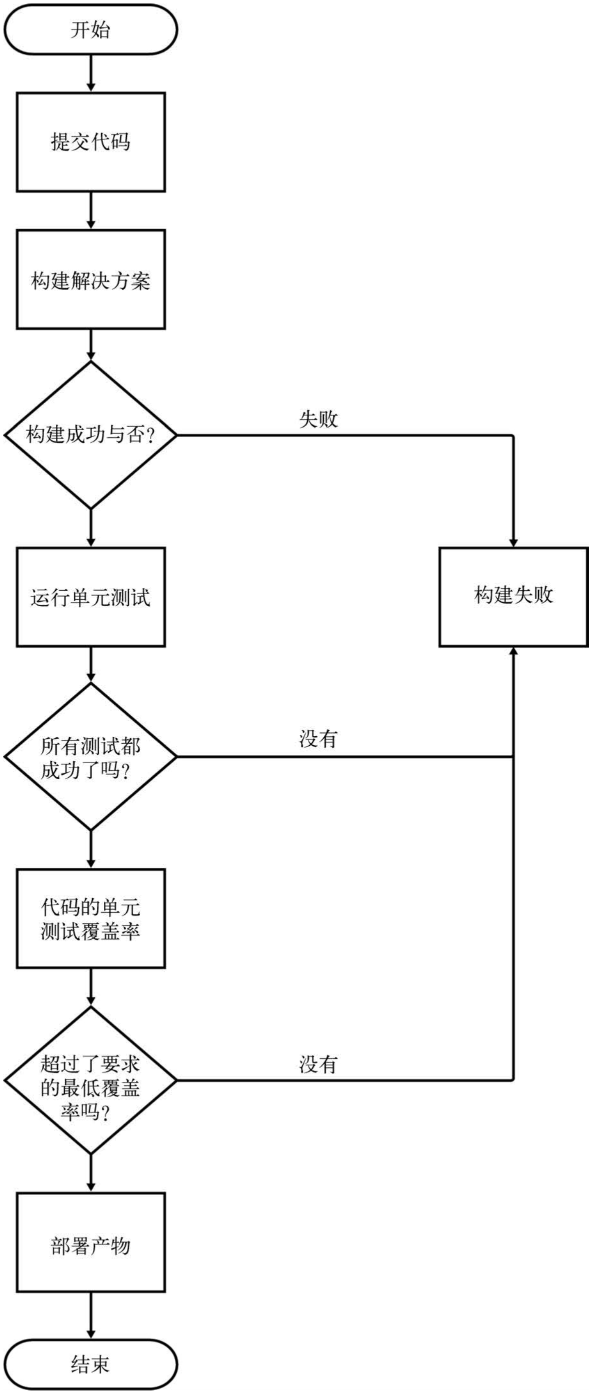
执行完这个命令后，所有远程的更新都合并到了本地，此时，本地分支和对应的远程分支是完全相同的。

A.3 持续集成

当开发人员把代码推送到中央存储库时，通常会在服务器编译最新的代码。这种对开发人员所做代码变更的持续集成能提供有关代码状态的非常有价值的及时反馈。如果编译失败了，就无法满足拉请求验收的第一个标准：导致无法构建可工作版本的任何请求都会被立刻拒绝。

然而，仅仅通过编译最新代码是无法确保开发人员为修复缺陷或实现新特性所做的变更是否影响了软件的其他部分。因此，在代码编译成功后，持续集成系统会继续运行所有单元测试以判断是否有其他已有的用例被破坏，通过所有单元测试后，系统还会继续检查代码的单元测试覆盖率是否满足要求的标准。完成所有这些步骤后，系统才会尝试从构建的输出中生成可供部署的安装包。

所有这些步骤是顺序执行的，每一步都成功是完成持续构建流程的前提条件。编译不成功就无需再运行单元测试；类似地，单元测试未通过就无需再检查代码的单元测试覆盖率；同样，单元测试覆盖率没有达到要求就无需生成可供部署的安装包。持续集成系统对每个推送的分支都执行这个过程能极大地减轻开发人员的负担。相比花费大量时间手动完成所有相关任务，有了持续集成系统，开发人员只需要在本地编译代码变更涉及的工程并运行针对这些变更编写的单元测试，持续集成系统会完成其余所有的工作。图A-3展示了上述这种持续集成构建流程的流程图。



图A-3　一个简单的持续集成服务的流程图

C#敏捷开发实践



用户的需求经常变化，每个开发者都深受其害。不过，如果能够提高代码的自适应性，就能更加轻松地响应变化，避免重复劳动。本书介绍了敏捷编程的最佳实践、原则和模式，能让你编写出灵活的自适应性代码，从而创造更大的商业价值。

专家指导，帮你跨越理论和实践之间的鸿沟

- 熟练运用Scrum：工件、角色、度量标准、阶段
- 组织和管理架构的依赖关系
- 回顾各种模式、反模式以及最佳实践
- 掌握SOLID原则
- 管理自适应代码的各种接口应用方式
- 先后进行单元测试和重构
- 观察委托和抽象如何影响代码自适应性
- 学习实现依赖注入的最佳方式
- 将学到的知识应用于敏捷开发的实际项目

Amazon读者推荐

“本书从C#的角度总结了过去十年间关于敏捷的主要话题，十分简洁、实用。代码示例贯穿始终，帮助程序员理解原则和模式背后的思想。与其他程序开发图书的最大不同是，这本书并不教条。”

“这本书对于所有层次的软件工程师来说都是一本不可多得的好书。它介绍了广为人知的软件设计原则，提供了如何将这些原则应用于现实软件产品中的实际示例。我想把此书推荐给想要掌握好C#语言的所有人。”

Microsoft

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/程序设计/C#

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-42789-2

ISBN 978-7-115-42789-2
定价：69.00元